

Cloud Computing

Virtual Resources and Distributed Cloud Applications

Chapter 5: Data-Intensive Applications



Distributed and Operating Systems
Electrical Engineering and Computer Science
Technische Universität Berlin

Overview

- **Introduction**
- Distributed Storage
 - GFS
 - HDFS
 - Bigtable
- Distributed Processing
 - MapReduce
 - Spark
 - Flink

Data-Driven Applications



lifecycle management



home automation



health



water management



market research



information
marketplaces



traffic management



energy management

- Trends: Digitalization, Internet of Things, Smart Cities

Data-Intensive Applications – Clouds (1/2)

- Requirements for data-intensive applications out-scaled traditional solutions
- Commercial parallel databases and HPC computers often also prohibitively expensive
- New distributed systems for commodity clusters
 - Both storage and computing across commodity nodes, which allows to keep computation close to the data
 - Individual servers are less reliable, so fault tolerance must be provided by distributed systems

Data-Intensive Applications – Clouds (2/2)

- Cloud computing (particularly IaaS) is a good fit for data-intensive applications
 - Looks like a large pool of cluster nodes to customer
 - Possibility of elastic scaling
 - No large upfront capital expenses
- However, there are also downsides
 - Virtualization overhead for I/O operations
 - No control over physical infrastructure
- Interesting platform for infrequent users and high variance in usage, while heavy users with a steady usage might benefit from dedicated clusters

Challenges of Large-Scale Data Processing

- Large clusters/clouds have 100s to 1000s of servers
 - Very high performance/throughput possible
 - But jobs must be written to take advantage of the massively parallel and distributed environment
- Writing efficient and robust parallel and distributed processing jobs is hard
 - Most developers are no experts in parallel programming, distributed systems, and data processing
 - Developers also do not want to care about concurrency and failures, but about extracting new info from data
- Needed: Suitable abstraction layers for developers

Requirements for distributed data processing systems

- Developers don't have to think about parallelization and distributed execution (and instead write sequential code independent of the parallelism at runtime)
 - System schedules and manages distributed tasks
- Developers don't have to think about load balancing
 - System distributes the work evenly across workers
- Developers don't have to think about fault tolerance
 - System detects failed nodes / executors
 - System re-executes any lost computations

Analytics Cluster Setup

- Analytics clusters often shared by multiple users and applications
- Shared via resource management and distributed file systems

Applications

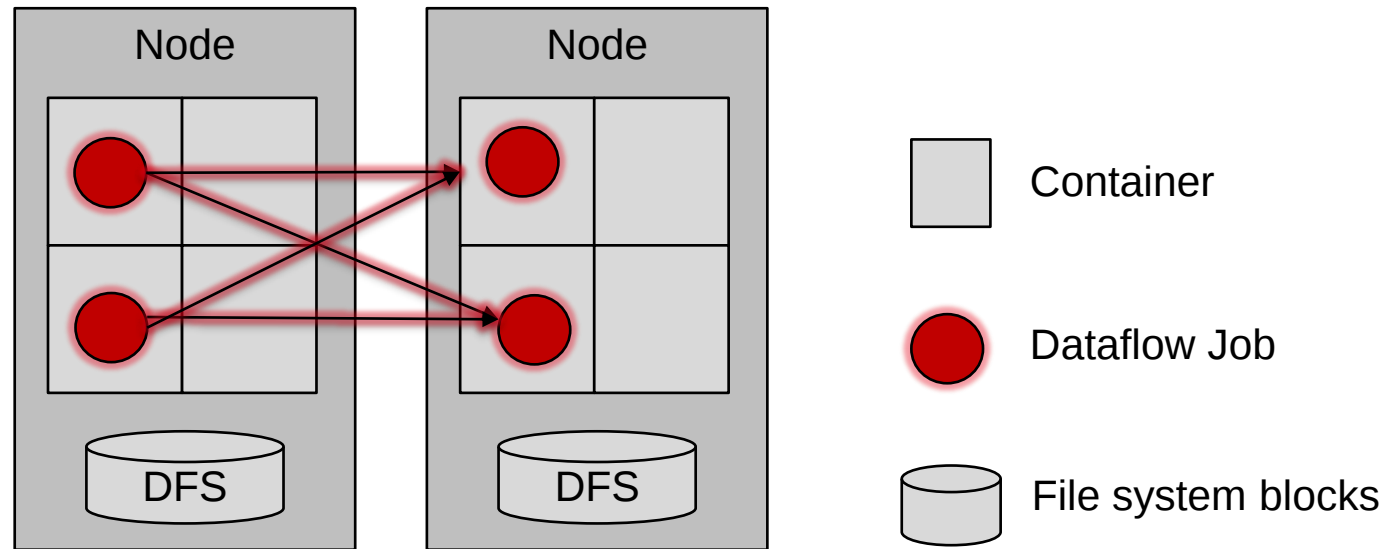
Processing Frameworks (e.g. Spark)

Resource Management System (e.g. YARN)

Distributed File System (e.g. HDFS)

Resource Management Containers

- Users reserve resources temporarily in “containers” from large sets of managed commodity nodes



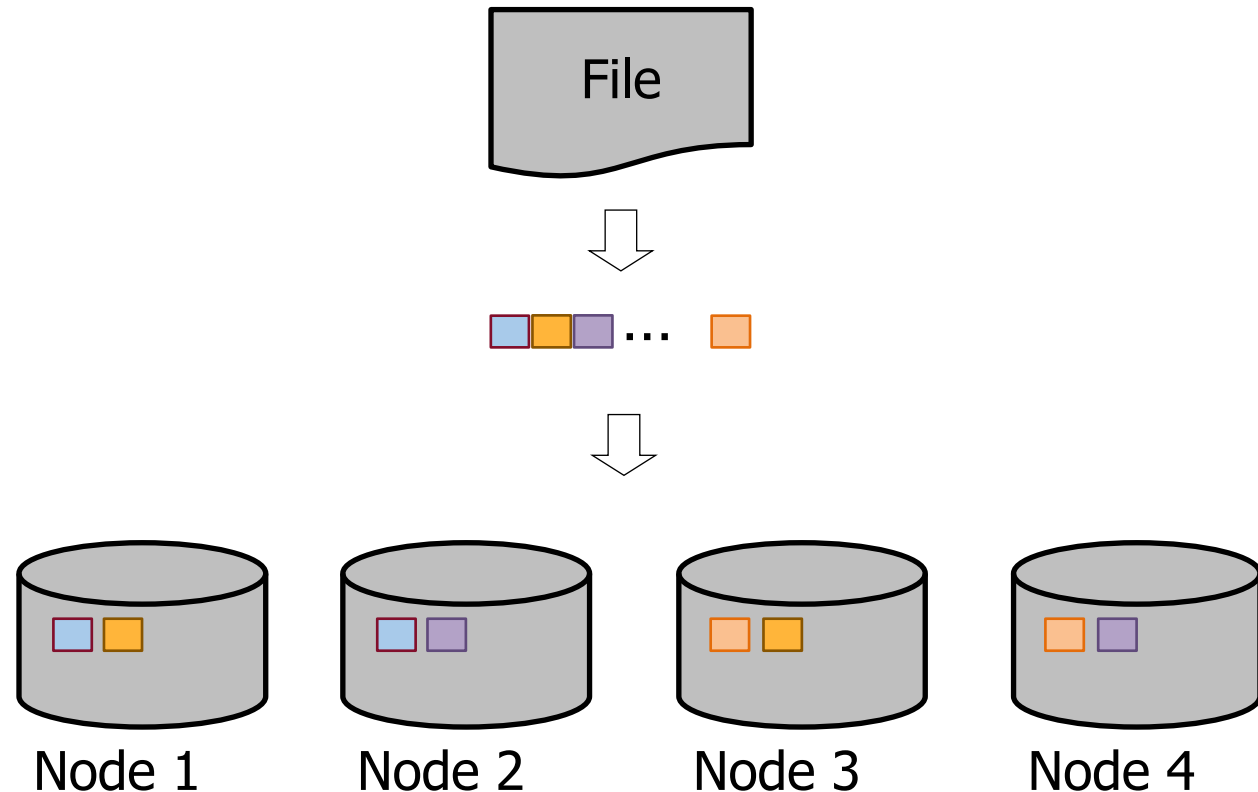
- Note: Here containers are a bundle of resources on a specific node assigned to a specific, usually distributed job, often without any isolation (and not necessarily OS-level containers)

Overview

- Introduction
- **Distributed Storage**
 - GFS
 - HDFS
 - Bigtable
- Distributed Processing
 - MapReduce
 - Spark
 - Flink

Distributed Data

- Distributed file systems and databases (e.g. Bigtable on GFS) typically running on commodity clusters
- Fault tolerance and parallel access through replication of *data blocks*



Overview

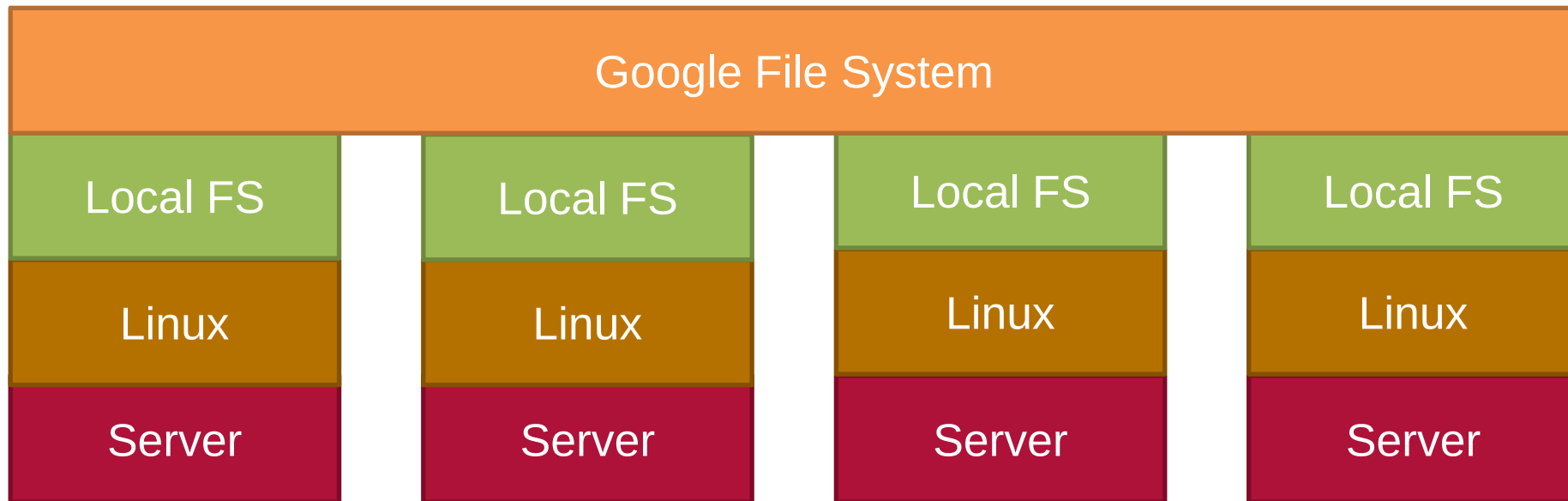
- Introduction
- Distributed Storage
 - **GFS**
 - HDFS
 - Bigtable
- Distributed Processing
 - MapReduce
 - Spark
 - Flink

Google File System[4]

- GFS was designed to meet the following criteria
 - Scalability: Must store hundreds of terabytes and support for large commodity clusters (1000s of servers)
 - High performance: High throughput (over low latency)
 - Fault tolerance: Compensate individual server failures
- Google's workload characteristics
 - Individual files are expected to be big (MBs to GBs)
 - Billions of files must be managed by the file system
 - Most files are written only once, then read sequentially

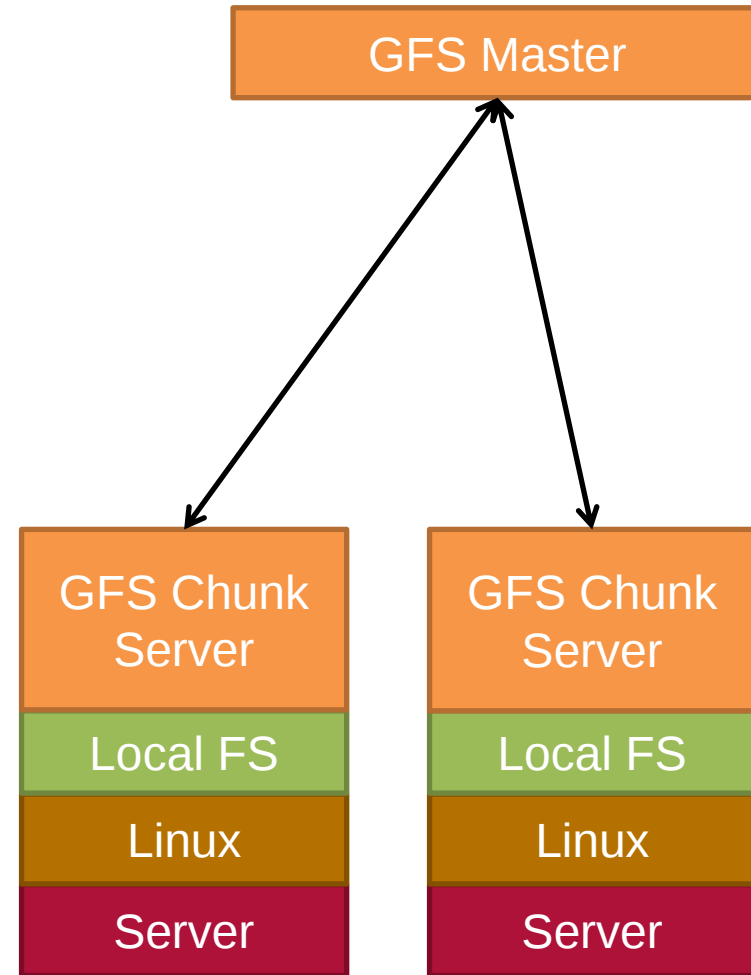
Google File System Design

- GFS is not a POSIX-compliant file system
 - Limited support for random writes (expected to happen rarely), data can be efficiently appended instead
 - Sits on top of regular file systems, but provides its own API to users



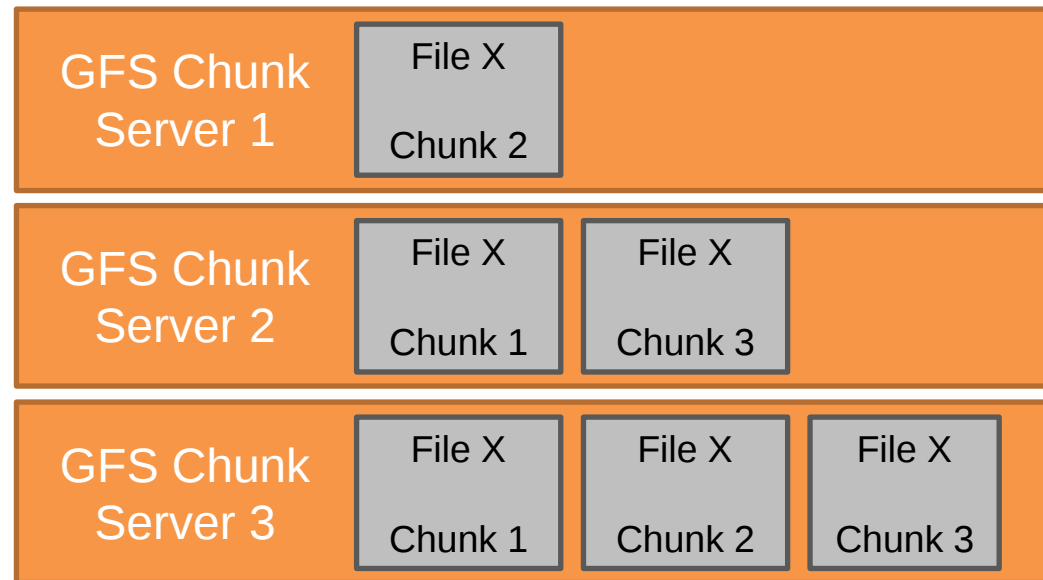
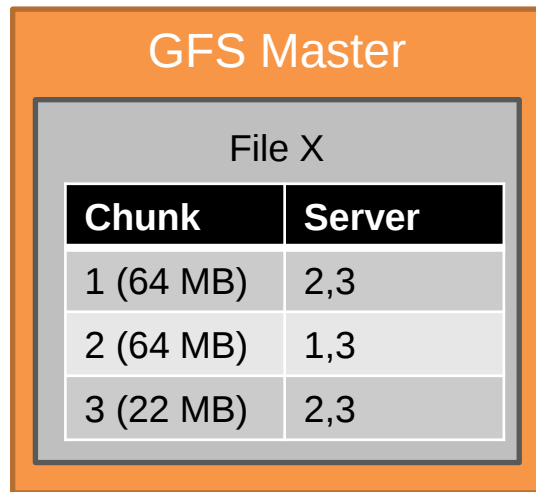
Google File System Architecture

- GFS follows master-worker architecture
 - Master stores metadata
 - Workers store actual data
 - ◆ Called chunk server
 - ◆ One chunk server per node
- Files are split into fixed-sized chunks
 - Identified by GUIDs
 - Replicated across machines



Example of Data Storage in Google File System

- Let's assume, we have file X with a size of 150 MB
 - Default chunk size is 64 MB
 - File is stored in three chunks
 - Chunks are stored across the available chunk servers



Reading File X From GFS

- Let's assume client wants to read X starting at MB 100
 1. Client knows chunk size, converts offset to chunk index
 2. Client contacts master with filename and chunk index
 3. Master returns chunk handle and list of replicas
 4. Client sends read request to closest replica
 5. Client caches chunk information so further reads of the same chunk require no more client-master interaction (until cached information expires)
- Note: Master not involved in actual data transfer

Design Considerations for the GFS Master

- Central master has global knowledge of the file system
 - Simplifies design and reduces response times
 - Easy garbage collection and data reorganization
- Metadata is kept in main memory for performance
 - Metadata for each chunk consumes 64 bytes
 - ➔ Chunk size is an important parameter of GFS (determines the amount of metadata that fits into memory and the amount of metadata exchanged between master and client)

What Happens When the Master Fails?

- Metadata is crucial to the functionality of the FS
- GFS knows three types of metadata
 - Namespace mappings (files, directories)
 - File-to-chunk mappings
 - Replica locations
- Namespace/file-to-chunk mappings are written to log
 - Operation log is persistent on master's hard disk
 - Also replicated to remote machine
- Replica locations are requested from chunk servers when the master (re-)starts or new chunk server joins

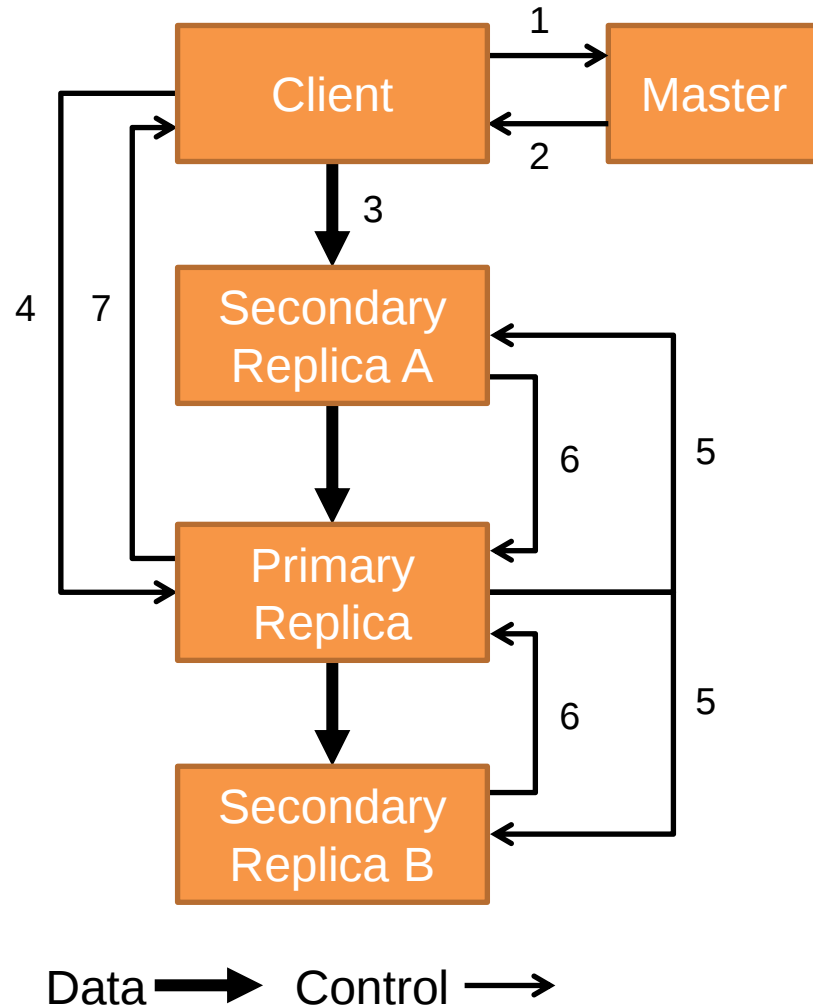
Operation Log

- Serves as logical timeline that defines the order of concurrent operations
 - Files and chunks (as well as their version) are all uniquely and eternally identified by their logical creation times
 - Changes to the file system are only made visible to clients after the log has been flushed locally and remotely
- Master can recover file system state by replaying log
- When log grows beyond certain size the master creates snapshot of metadata to improve startup times

GFS Leases and Mutation Order

- A mutation (write/append) is performed at all the chunk's replicas
 - Master grants chunk lease to one replica to ensure consistent mutation order across all other replicas
 - Replica with lease is called primary
 - Primary chooses serial order for mutations
 - All other replicas follow this order
- Global mutation order defined by selected primary

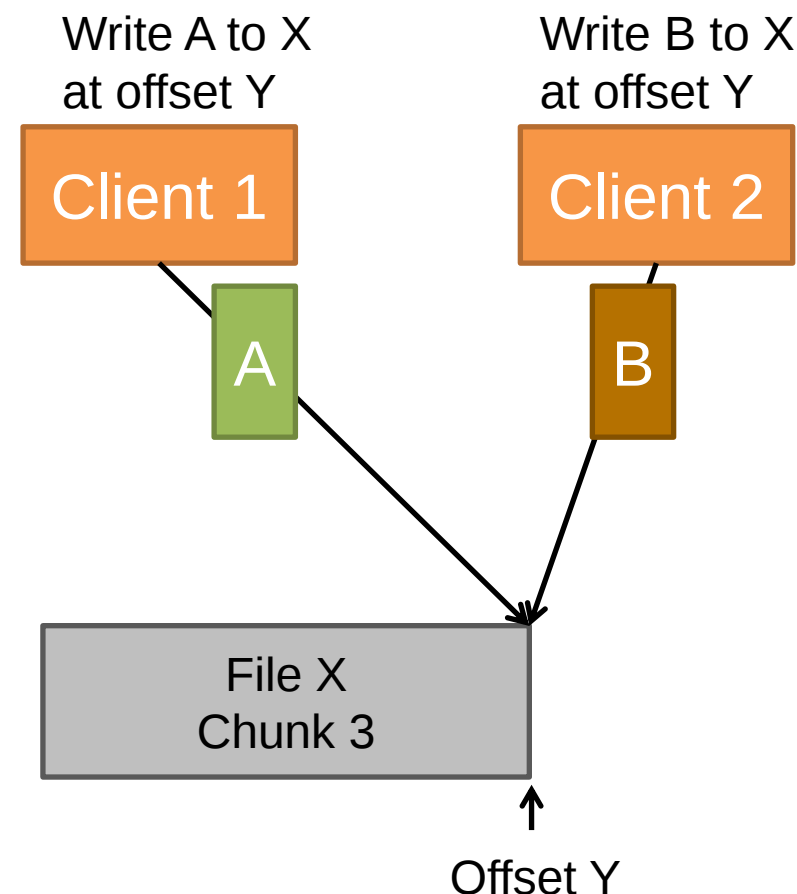
GFS Control/Data Flow for Writes/Appends



1. Request primary, list of replicas from master
2. Master's response
3. Data transfer to arbitrary replica server
4. Mutation request to primary, primary determines order
5. Replicas apply mutations in primary's serial order
6. Ack. to primary
7. Ack. to client

GFS Support for Atomic Appends (1/2)

- Technically, append is writing to a file at specific offset
 - Offset for reads specified by client
 - Challenging for writes in presence of concurrency
 - ◆ Example: Two applications concurrently specify to write record at offset Y
 - ◆ One record would be destroyed
 - Applications depend on atomic append for no loss of data



GFS Support for Atomic Appends (2/2)

- Special GFS operation to append data in atomic units
 - Append procedure
 1. Client specifies append operation and data (but no offset!)
 2. Primary serializes requests, chooses offsets
 3. Returns offsets to clients after data has been appended
 - Data appended to next chunk if size exceeded otherwise
 - Rest of previous chunk filled with padding bits!
 - Client retries operation if append fails at any replica
 - Appended data may occur more than once in some replicas
 - Replicas are not guaranteed to be bitwise identical!
 - Operation guarantees that data is appended *at least once*

Overview

- Introduction
- Distributed Storage
 - GFS
 - **HDFS**
 - Bigtable
- Distributed Processing
 - MapReduce
 - Spark
 - Flink

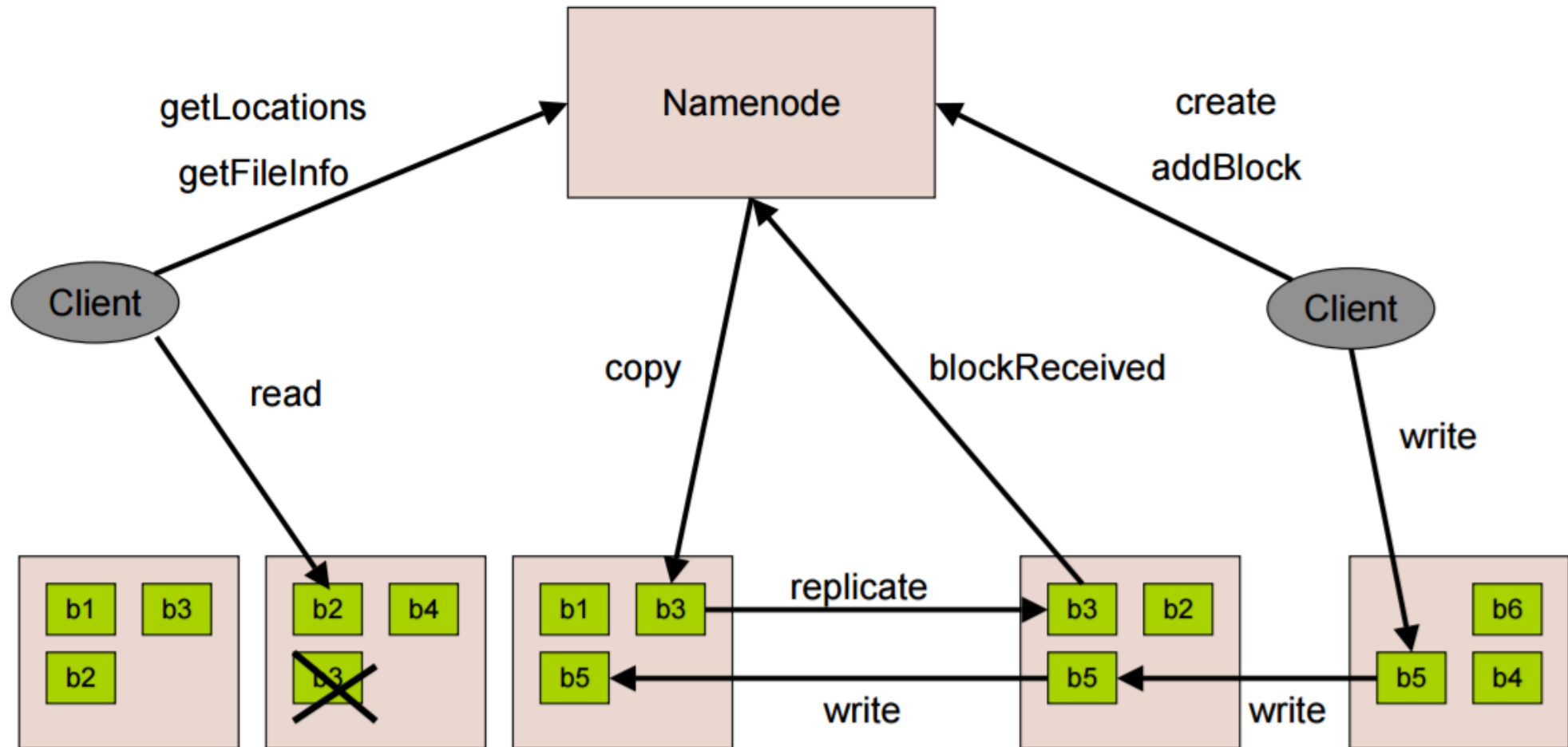
Hadoop Distributed File System (HDFS) (1/2)

- Distributed filesystem inspired by GFS: All data is stored in blocks, replicated on multiple nodes
- Each node is a Linux compute node with hard disks
- One NameNode maintains the file locations, many DataNodes store actual data (1000+)
- Any piece of data is available if at least one data node with a replica is up and running
- Rack-optimized by default: one replica written locally, second in same rack, third replica in another rack
- Uses large block size, 128 MB is the default, designed for batch processing

Hadoop Distributed File System (HDFS) (2/2)

- Optimized for scalability and “write once, read often”
- Applications need to be re-engineered to only do sequential writes
- Example systems working on top of HDFS
 - HBase (Hadoop Database), a database system with only sequential writes, inspired by Google Bigtable
 - MapReduce, a batch processing system
 - Pig and Hive, data mining tools
 - Mahout, a set of machine learning libraries
 - Spark and Flink, dataflow systems

HDFS Architecture



Hadoop Distributed File System (HDFS)

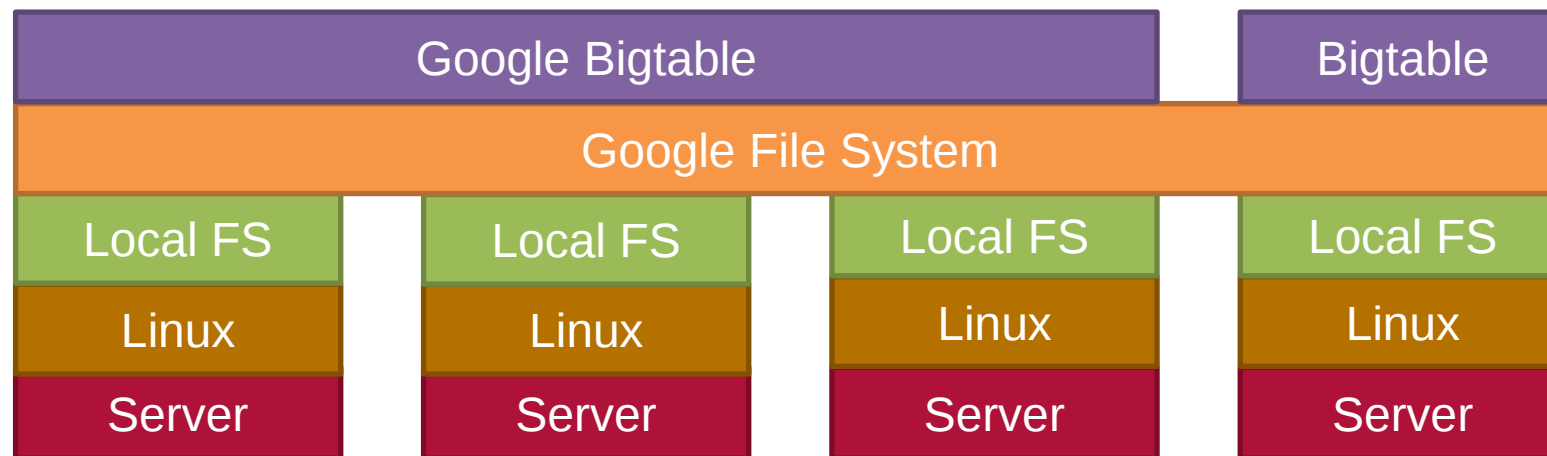
- NameNode is a single computer that maintains the namespace (metadata) of the filesystem
 - Keeps all metadata in memory, writes logs, and does periodic snapshots to the disk
 - High Availability HDFS allows for multiple NameNodes
- Rules
 - Replica writes are done in a pipelined fashion and reads are done from the nearest replica to optimize network usage
 - Replication and block placement: file's replication factor can be changed dynamically (default is 3)
 - Block placement is rack-aware (by default, when racks are configured)

Overview

- Introduction
- Distributed Storage
 - GFS
 - HDFS
 - **Bigtable**
- Distributed Processing
 - MapReduce
 - Spark
 - Flink

Google Bigtable[5]

- GFS offers enormous storage capacities, but limited support to efficiently retrieve structured data (== no indices)
- Solution: Google Bigtable
 - High-throughput NoSQL store (wide column store), through multi-dimensional sorted map, implemented on top of GFS
 - Many use cases at Google, e.g. Web indexes, Gmail, Maps
 - Open-source clones: Hbase on HDFS and Cassandra

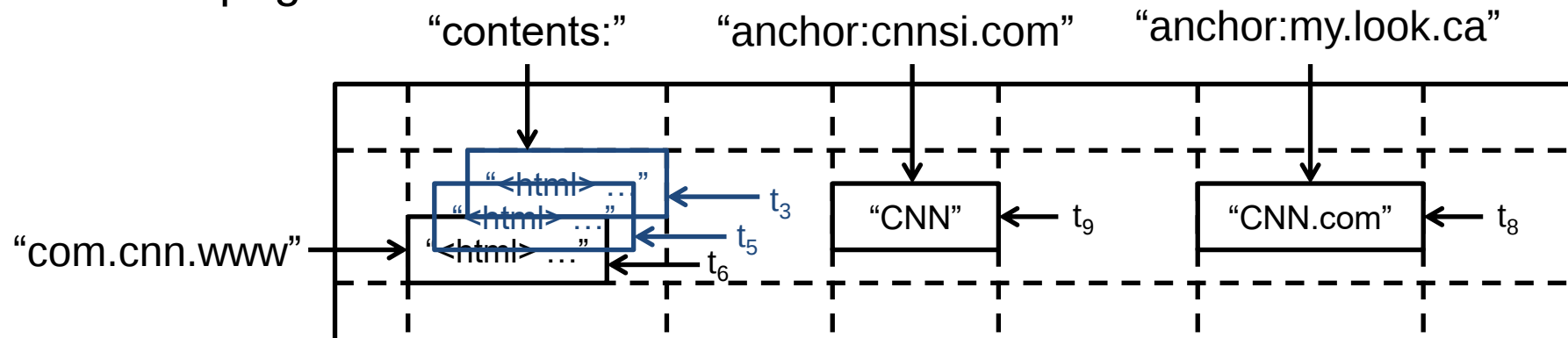


Google Bigtable[5]

- Sequential writes are much faster than random writes
- Data is stored sorted by the row key and sharded (split to different servers)
 - Allowing for efficient scans in increasing alphabetical row key order
- Columns are grouped in “column families” and all columns in a single column family are stored together in compressed form on disk
 - Some queries might not access all columns → column families can be easily skipped
- Timestamp field keeps track of versions, e.g. Website content changes over time

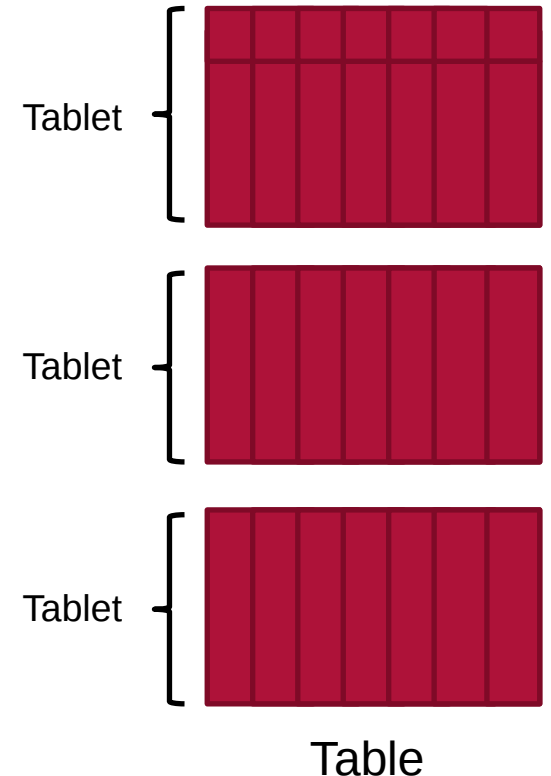
Bigtable Data Model

- (row:string, column:string, time:int64) → string
- No schema, each row can have arbitrary columns
- Columns are grouped to column families (on disk). Example:
 - Row key = reverse of URL to group pages from the same site
 - “Contents” column family stores Web site contents (certain version)
 - “Anchor” column family stores links pointing to the Web page
 - Easy to skip for example reading the Web page contents if we are only interested in links between Web pages



Bigtable Terminology

- Bigtable clusters store several tables
- Rows with consecutive row keys are grouped together to “tablets” = unit of distribution and load balancing
- A table consists of a set of tablets
 - Tablet contains all data associated with a row range
 - Initially, each table has one tablet
 - When tablet grows, it is automatically split into several tablets
 - ◆ Avg. tablet size is 100-200 MB
 - ◆ Servers typically hold about 100 tablets

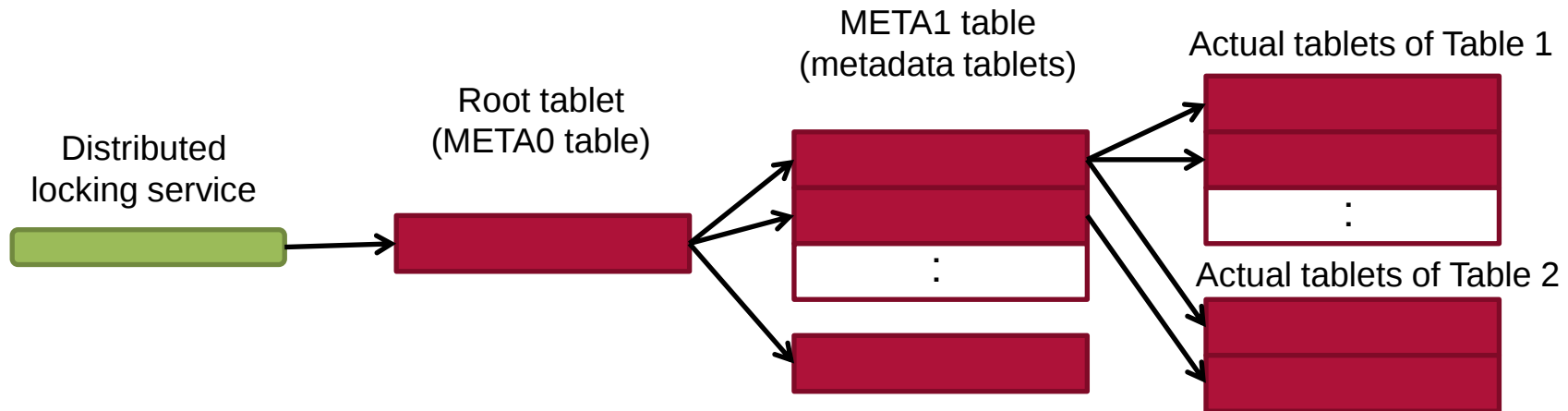


Bigtable Architecture

- Client
 - Sends requests (row, column, time), expects responses (values)
- Master
 - Can manage several tables
 - Assigning tablets to tablet servers
 - Keeps track of addition/expiration of tablet servers
 - Load balancing and garbage collection
- Tablet servers
 - Store the actual tablets
 - Serve client requests
 - ◆ Clients rarely communicate with the master

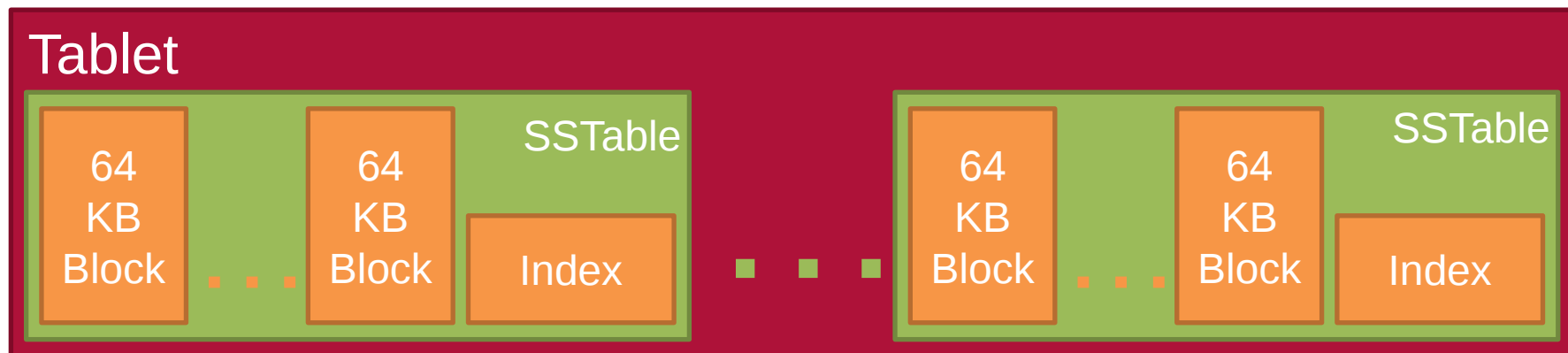
Locating Tablets

- Tree to locate correct table
 - Client specifies table and row key
 - First lookup in root tablet (pointer in dist. locking service)
- Information is stored in special METADATA tables
 - Each row in METADATA table consumes 1 KB
 - 2^{34} tablets addressable (assuming 128 MB tablet size)



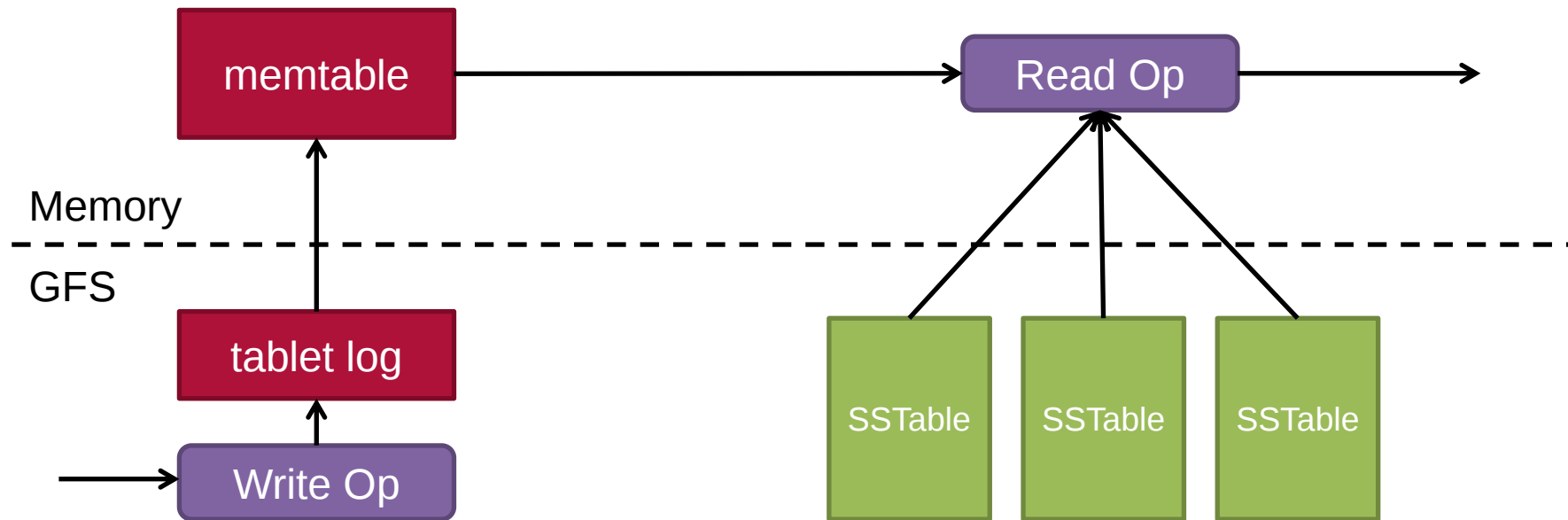
Internal Tablet Structure

- Actual data of a tablet is stored in a structure called SSTable (Sorted String Table)
 - Persistent, ordered, immutable map
 - Each SSTable is a file in GFS
- Each SSTable consists of several blocks and indices
 - Each block has 64 KB size
 - Indices loaded in main memory when SSTable is loaded
 - Blocks can be loaded with a single disk seek



Serving Tablets

- Updates are stored to a commit log in GFS
 - When committed they are stored in a memtable
 - Older updates are stored in a sequence of SSTables
- Reads are served by merging memtable and SSTables



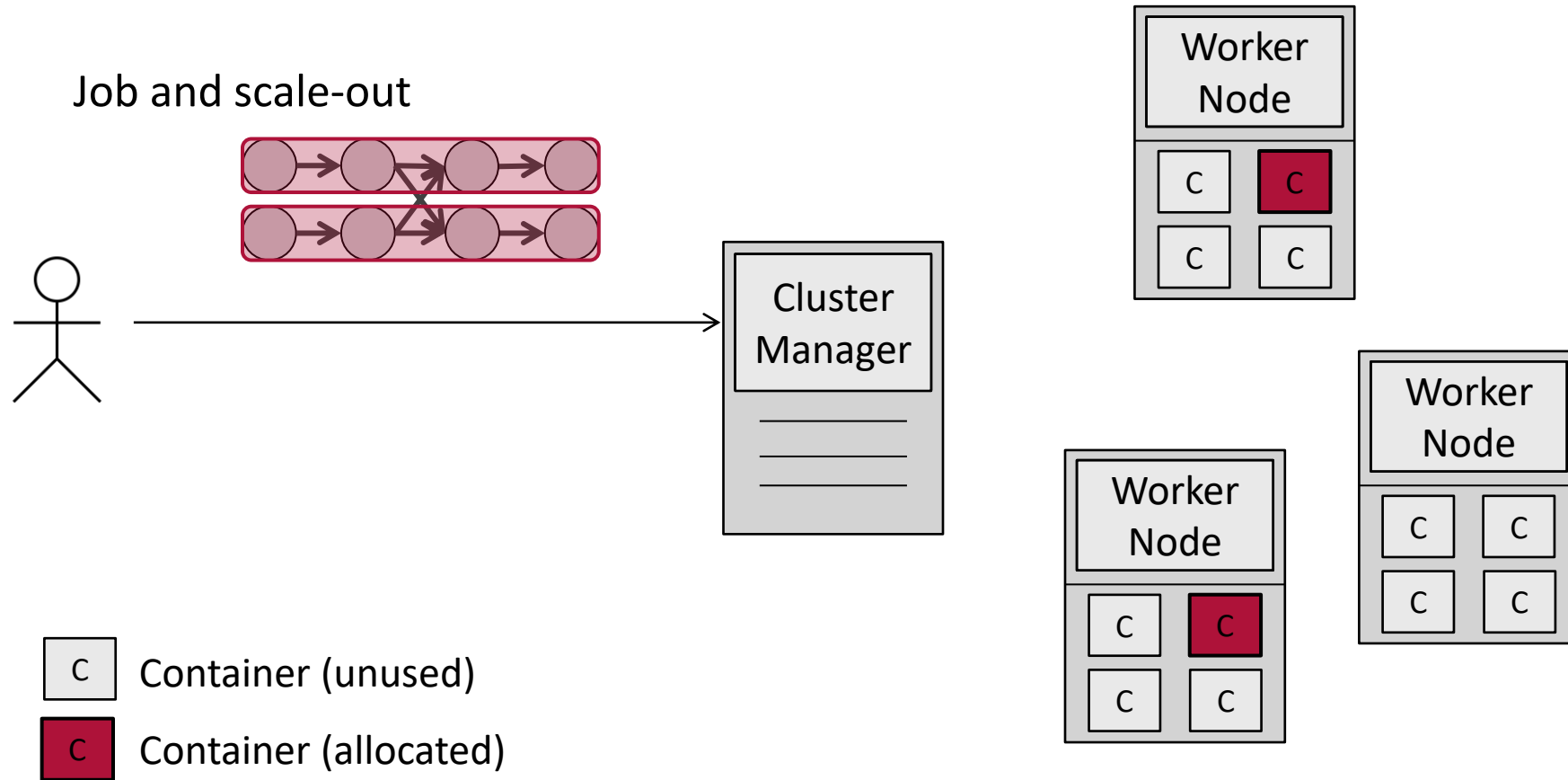
Compaction

- Minor compaction: Convert memtable into an SSTable
 - Reduces memory usage
 - Reduces reconstruction effort of memtable on restart
- Merging compaction
 - Reduces number of SSTables
 - Happens periodically in the background
- Major compaction
 - Merging compaction that results in only one SSTable
 - Special case of merging compaction that purges deleted data

Overview

- Introduction
- Distributed Storage
 - GFS
 - HDFS
 - Bigtable
- **Distributed Processing**
 - MapReduce
 - Spark
 - Flink

Distributed Execution on Commodity Nodes



Overview

- Introduction
- Distributed Storage
 - GFS
 - HDFS
 - Bigtable
- Distributed Processing
 - **MapReduce**
 - Spark
 - Flink

MapReduce[2]



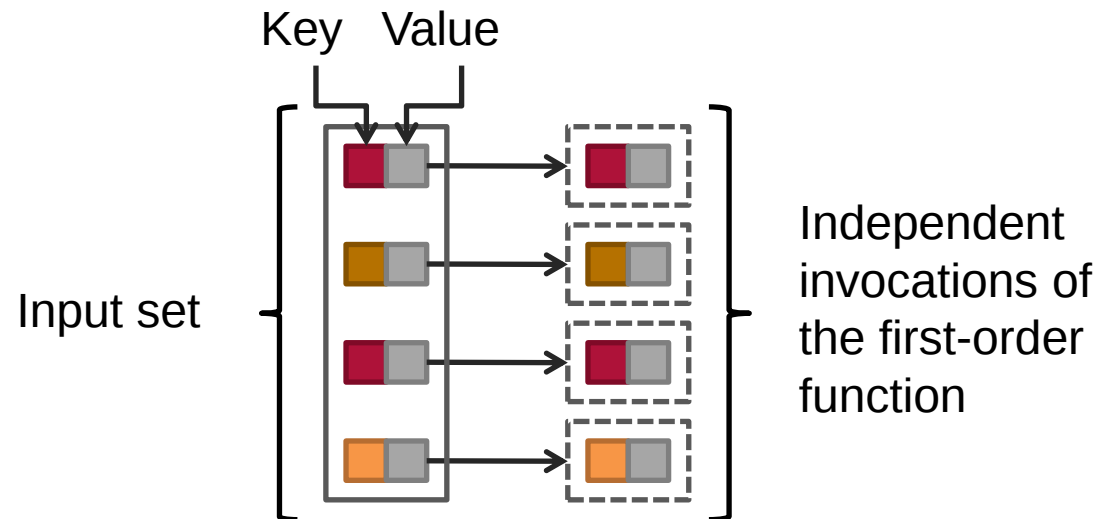
- System published by Google in 2004
 - Introduces MapReduce programming model
 - Illustrates how model helps to meet the discussed requirements (sequential code, fault tolerance, ...)
- Google used MapReduce for various things
 - Process crawled documents, logs
 - Computing inverted indices
 - ...
- Publication has spawned significant research activities
 - Among the most cited CS research papers in last 15 years
- System has been replaced by successors by now

The MapReduce Programming Model

- Based on second-order functions *map* and *reduce*
 - Inspired by functional programming language Lisp
- *Map* and *reduce* take first-order functions as input
 - Specify signature of the first-order function
 - Specify how data is passed to first-order function
- MapReduce operates on a key-value (KV) model
 - Data is passed as KV pairs through the system
 - Developers specify how to build KV pairs from input data

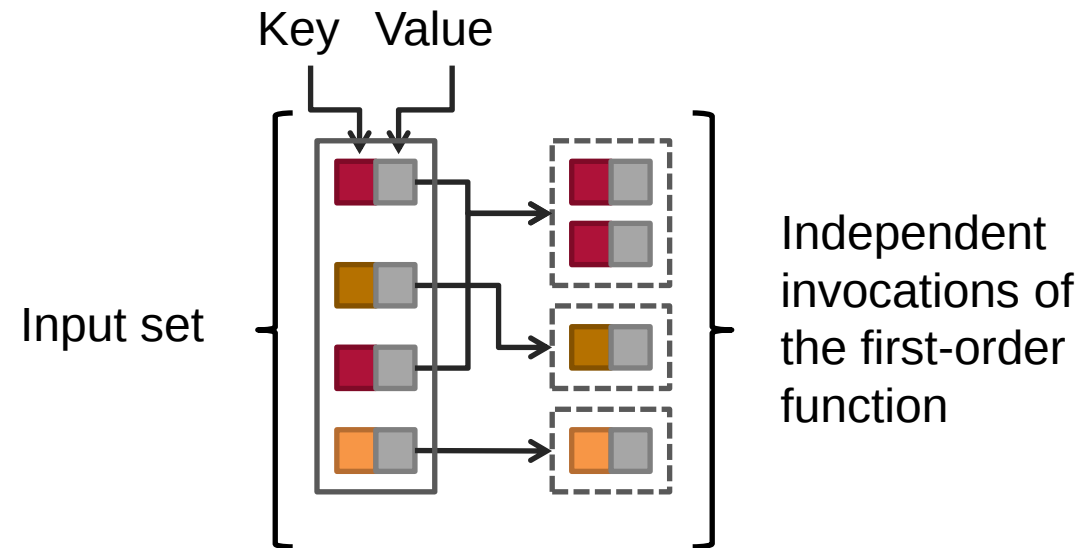
The Map Function

- Signature: $(k1, v1) \rightarrow \text{list}(k2, v2)$
- Guarantees to the first-order function:
 - First-order function is invoked once for each KV pair
 - Can produce a $[0, *]$ KV pairs as output
- Useful for projections, selection, ...

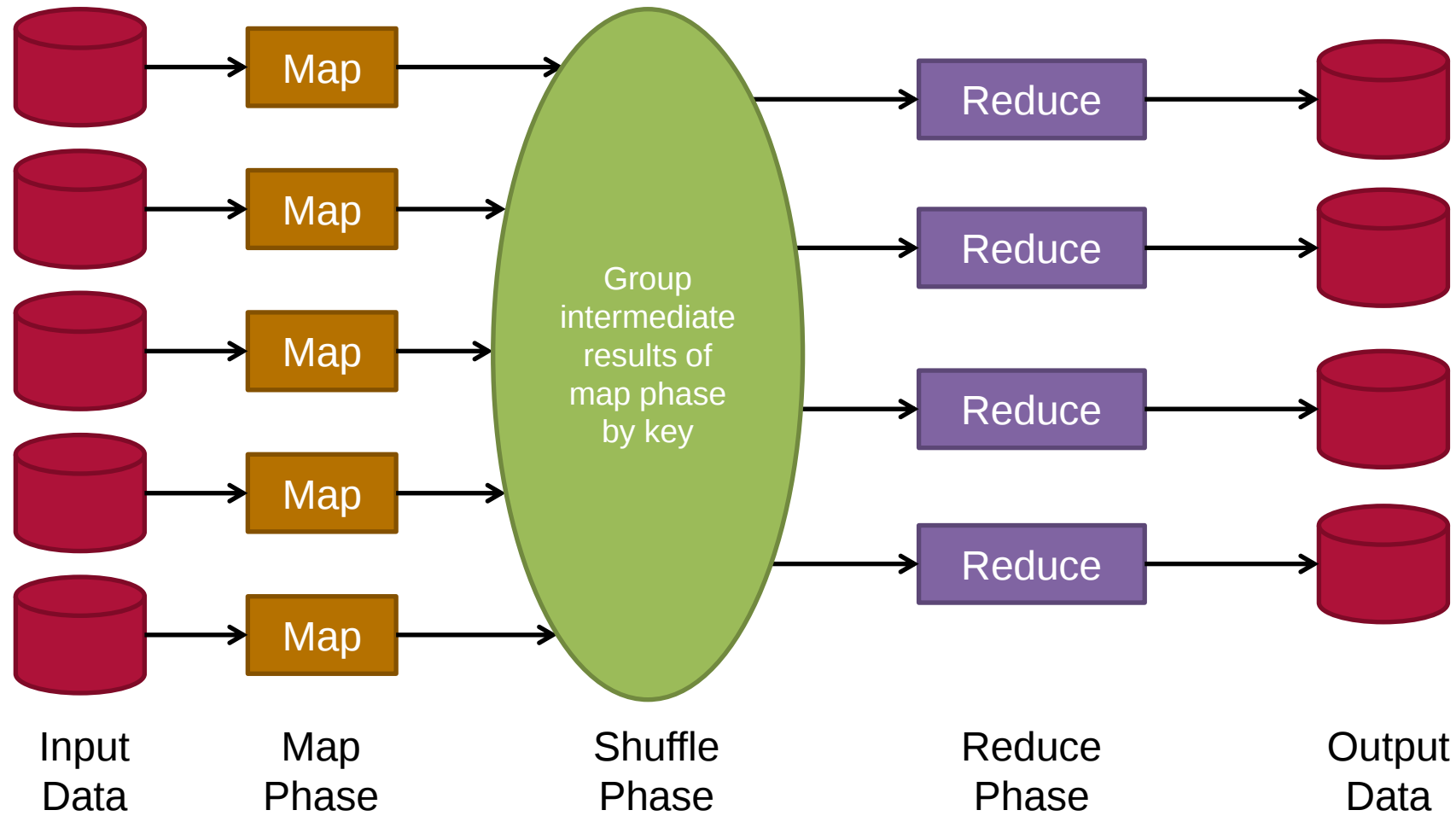


The Reduce Function

- Signature: $(k2, \text{list}(v2)) \rightarrow \text{list}(k3, v3)$
- Guarantees to the first-order function:
 - All KV-pairs with the same key are presented to the same invocation of the first-order function
- Useful for aggregation, grouping, ...



High-Level View on a MapReduce Program



MapReduce Example: Word Count (1/4)

- Task: Count the occurrences of words in a text
- Example text:

1:	One For The Money
2:	Two For The Show
3:	Three To Get Ready
4:	Now Go Cat Go

- KV pairs for the map function: <Line number, line>
 - <1, One For The Money>, <2, Two For The Show>, ...
 - Four KV pairs
 - Four invocations of the map function

MapReduce Example: Word Count (2/4)

- Code of Word Count map function:

```
Map(long lineNumber, string line) {  
    for each word w in line {  
        EmitIntermediate(w, 1);  
    }  
}
```

- Intermediate results produced by map functions
 1. <One, 1>, <For, 1>, <The, 1>, <Money, 1>
 2. <Two, 1>, <For, 1>, <The, 1>, <Show, 1>
 3. <Three, 1>, <To, 1>, <Get, 1>, <Ready, 1>
 4. <Now, 1>, <Go, 1>, <Cat, 1>, <Go, 1>

MapReduce Example: Word Count (3/4)

- In the shuffle phase the intermediate results from the map invocations are grouped by key

→ Input for the reduce phase

1. <Cat, (1)>
2. <For, (1, 1)>
3. <Get, (1)>
4. <Go, (1, 1)>
5. <Money, (1)>
6. <Now, (1)>
7. <One, (1)>

8. <Ready, (1)>
9. <Show, (1)>
10. <The, (1, 1)>
11. <Three, (1)>
12. <To, (1)>
13. <Two, (1)>

MapReduce Example: Word Count (4/4)

- Code of Word Count reduce function:

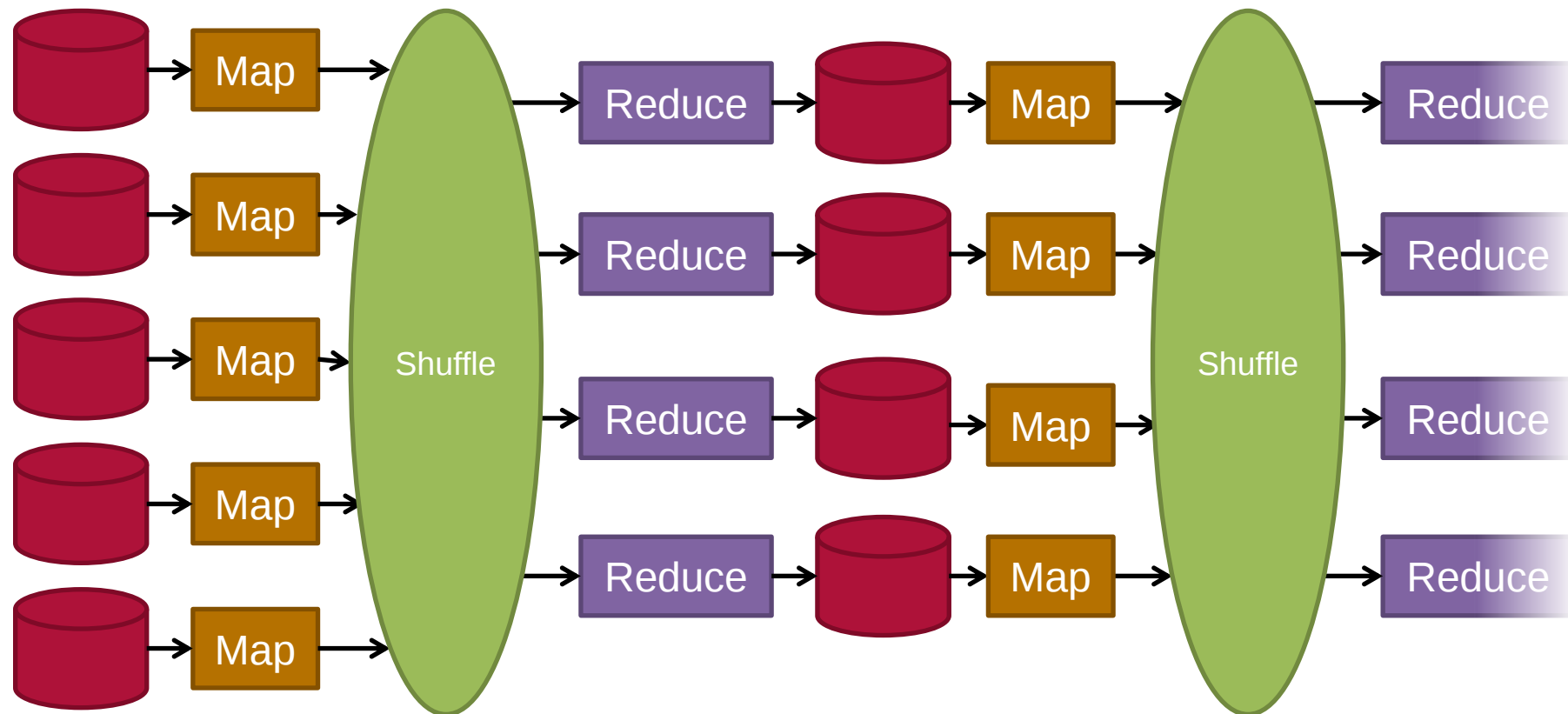
```
Reduce(String word, Iterator partialCounts) {  
    int result = 0;  
    for each v in partialCounts {  
        result +=v;  
    }  
    Emit(word, result);  
}
```

- Final results after the MapReduce job in output file

1. <Cat, 1>	5. <Money, 1>	9. <Show, 1>
2. <For, 2>	6. <Now, 1>	10. <The, 2>
3. <Get, 1>	7. <One, 1>	11. <Three, 1>
4. <Go, 2>	8. <Ready, 1>	12. <To, 1>
		13. <Two, 1>

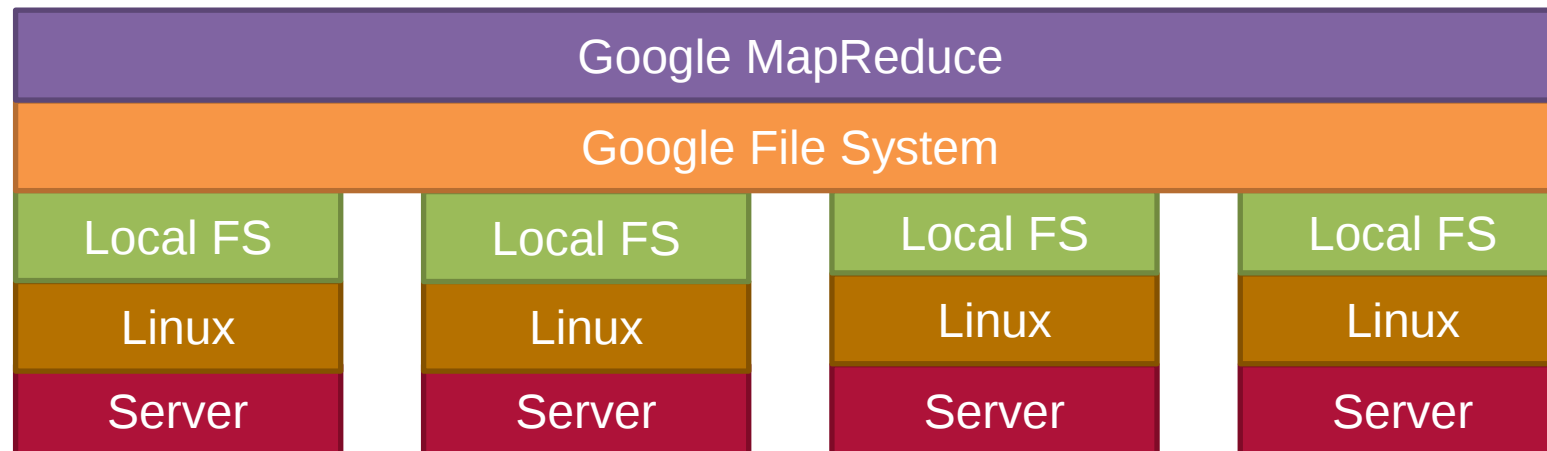
Combining Multiple MapReduce Programs

- More complex programs can be created by combining individual MapReduce jobs



MapReduce Implementation – Assumptions

- Designed for large sets of shared-nothing nodes
- Expects distributed file system
 - Every node can potentially read every part of the input, but local data is preferred (computation pushed to data)
- Individual nodes can fail



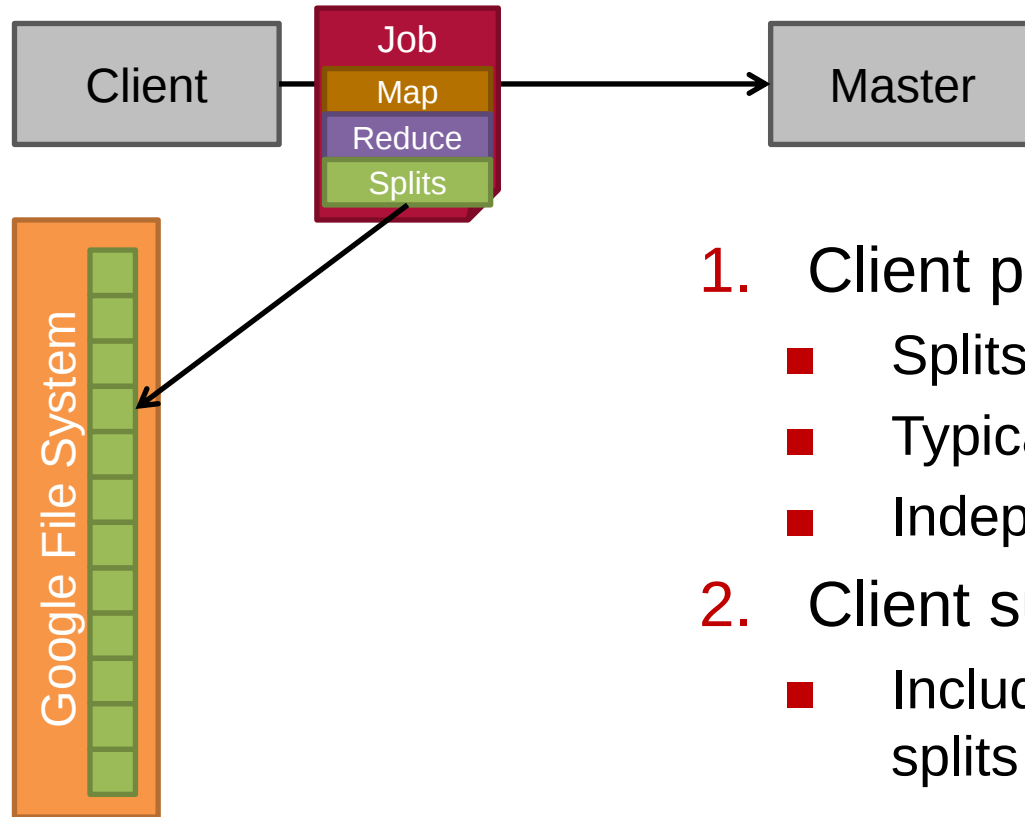
MapReduce Implementation – Design

- MapReduce follows master-worker pattern
- Master
 - Job scheduling
 - Load balancing
 - Monitoring of worker nodes and failure handling
- Workers
 - Execution of map and reduce functions
 - Reading and storing of data
 - Periodically reporting availability to master node

Some MapReduce Terminology

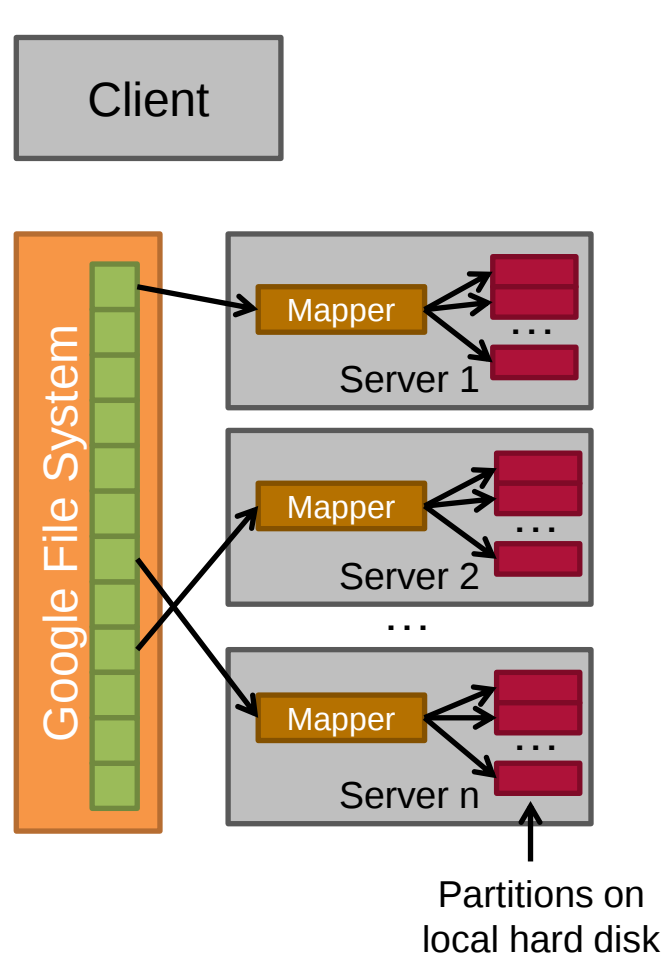
- Map function
 - User-defined first-order function that specifies what happens to the data in a job's map phase
- Mapper
 - Worker process invoking map function on each KV pair
- Reduce function
 - User-defined first-order function that specifies what happens to the data in a job's reduce phase
- Reducer
 - Worker process invoking reduce function on KV pair groups

Distributed Execution of MapReduce Program (1/3)



1. Client partitions input file into input splits
 - Splits define maximal scale-out
 - Typical split size is 16-64 MB
 - Independent of GFS block size
2. Client submits job to master
 - Includes code of map/reduce functions and list of input splits (file boundaries)
 - Master tries to find free resources to schedule mappers/reducers

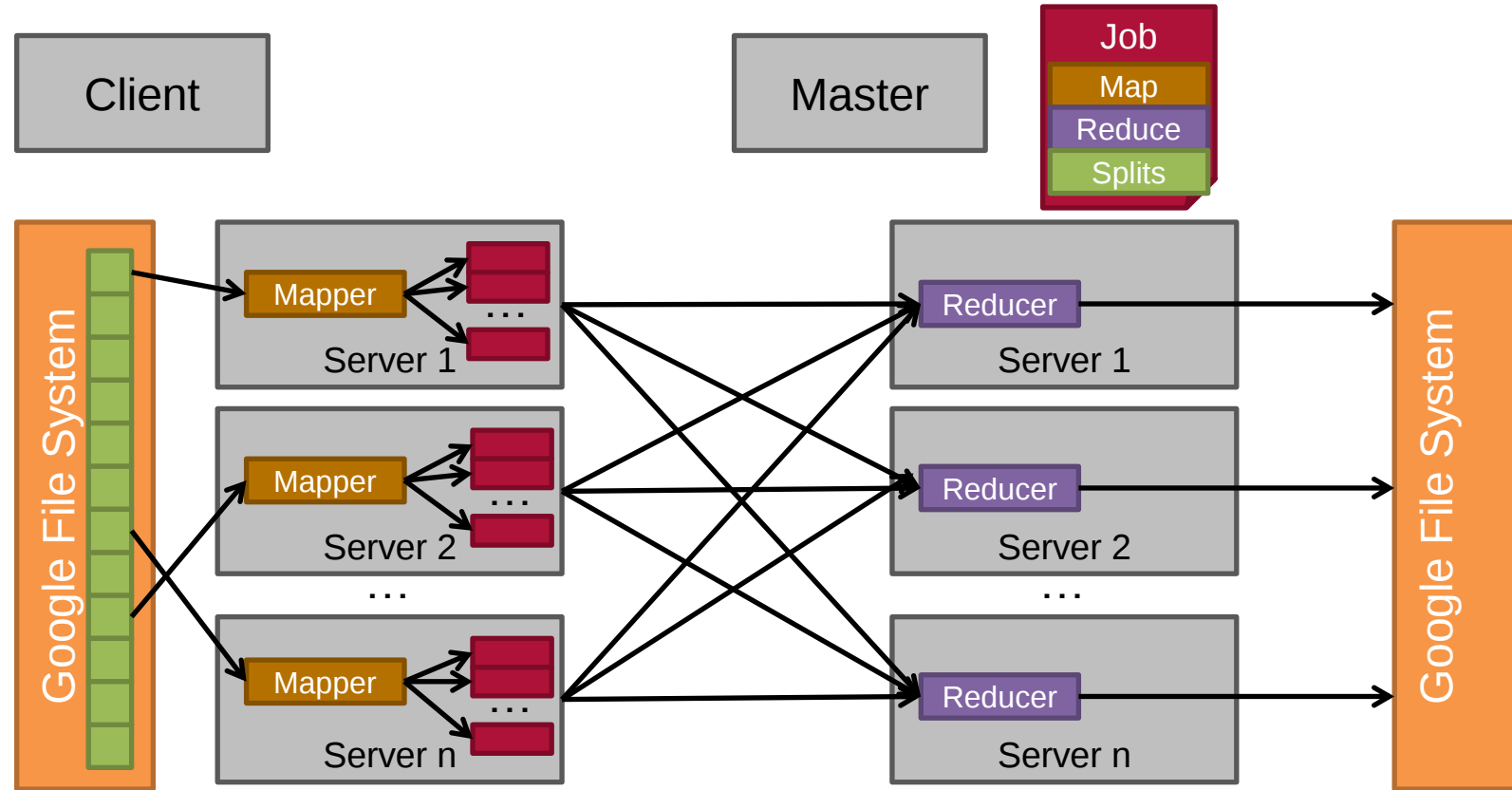
Distributed Execution of MapReduce Program (2/3)



3. Mapper started for each input split

- Executing the user's map function on each KV pair in input split
- Multiple mappers per server possible
- Servers with locally available replicas preferred (data locality)
- Intermediate output of mappers is partitioned by intermediate key, stored on server's local hard disk

Distributed Execution of MapReduce Program (3/3)



3. Reducers pull data from mappers over network: Data is grouped by key, passed to user's reduce function

MapReduce Refinement: Combiners

- Can be used to locally aggregate intermediate results
 - Executed after the mappers, before partitions go to disk
 - Useful to unburden network
 - Example: If intermediate KV pair $\langle \text{For}, 1 \rangle$, $\langle \text{For}, 1 \rangle$ is created by the same mapper, the combiner aggregates it to $\langle \text{For}, 2 \rangle$, without combiner $\langle \text{For}, (1,1) \rangle$ is shipped
- Applicability of combiner depends on job
 - Sometimes local aggregation destroys correctness of results
 - ◆ Example: Reduce function shall compute a median

MapReduce Fault Tolerance

- Scenario 1: Mapper process fails
 - Master detects failure through missing status report
 - Mapper is restarted on different node, continues with reading from GFS
- Scenario 2: Reducer process fails
 - Again, detected through missing status report
 - Reducer is restarted on different node, pulls intermediate results for its partition from mappers again
- Scenario 3: Entire worker node fails
 - Master re-schedules lost mappers and reducers
 - Already finished mappers may be restarted to re-compute lost intermediate results

Apache Hadoop

- Open-source implementation of Google's MapReduce
 - Written in Java
 - Many prominent users, e.g. Yahoo, Facebook, Twitter
- Also includes HDFS and YARN (a resource management system)
- Today, HDFS and YARN are more relevant than MapReduce as they are used with MR successors
- Hadoop still basis of many commercial distributions
 - e.g. Cloudera, Hortonworks, ...



MapReduce Limitations

- MapReduce is powerful but has two major limitations
 1. Assumes finite input (batch processing only)
 2. Data between stages and jobs goes to file system for fault tolerance
- Limitation of finite input prevents stream processing
 - Difficult to respond to events without large delays, e.g. click streams, IT systems monitoring ...
- Constraint to write to GFS especially costly for iterative algorithms
 - e.g. graph processing, machine learning, ...

Successors of MapReduce

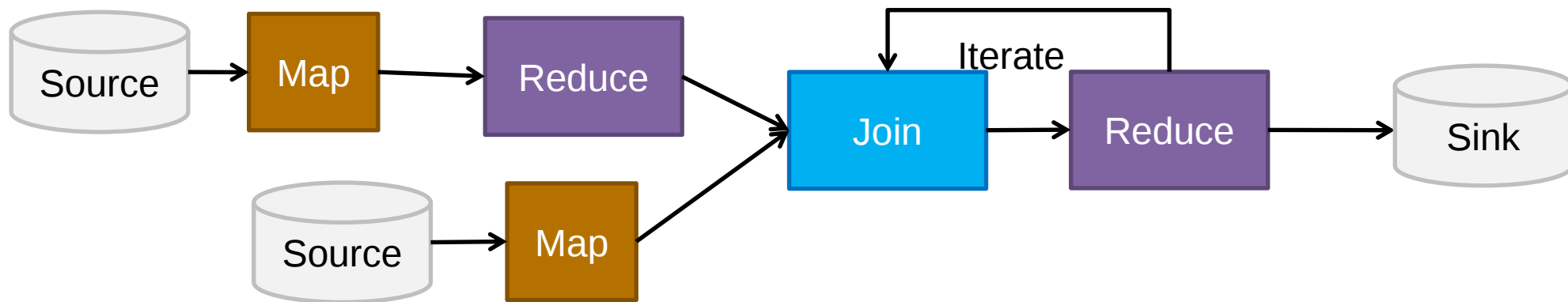
- MapReduce/Hadoop spawned several 2nd generation data processing frameworks
- 2nd generation of frameworks address the shortcomings of classic MapReduce



samza

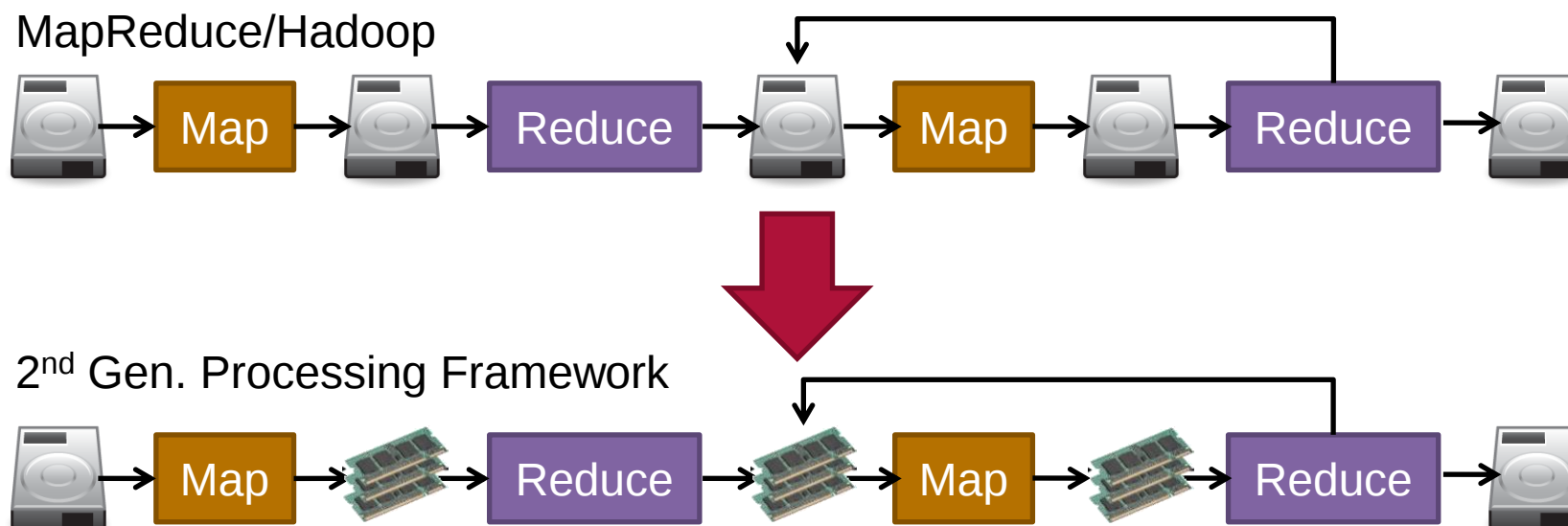
MapReduce Evolutions (1/3)

- Improved programming model with rich operator and feature set
 - Joins, groupBys, filters, iterations, and windows besides map and reduce operators
- This simplifies a wide array of computations, including:
 - Relational batch jobs
 - Iterative machine learning and graph processing
 - Continuous stream processing



MapReduce Evolutions (2/3)

- Intermediate results kept in memory
 - Instead of writing them to disk
 - Can speed up iterative programs by 100x



MapReduce Evolutions (3/3)

- Generalized processing engine that handles streams
 - Using a batch engine but small batches
 - ◆ Also called “Microbatching”
 - ◆ Creates “illusion” of continuous processing
 - Using a real streaming engine
 - ◆ Continuously deployed dataflows
 - ◆ Batch becomes a special case of streaming
- Stream processing requires re-definition of operators
 - e.g. classic reduce function does not make sense
 - Instead: window operators (e.g. all tuples of last 5 sec.)

Overview

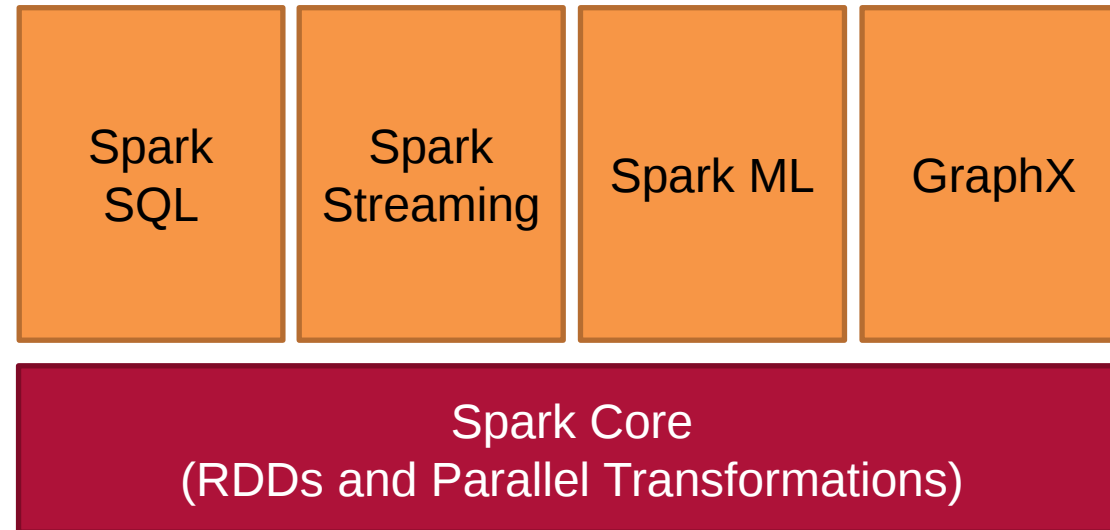
- Introduction
- Distributed Storage
 - GFS
 - HDFS
 - Bigtable
- Distributed Processing
 - MapReduce
 - **Spark**
 - Flink

Apache Spark_[7]

- General purpose 2nd generation data processing framework with support for in-memory datasets and processing
- Open source project, started as research project at UC Berkeley
- Apache top-level project since the beginning of 2014
 - Large user and developer base
 - Part of the platforms of Cloudera, Hortonworks, and IBM



Apache Spark Stack



- At Spark's core are parallel transformations of Resilient Distributed Datasets (RDDs)
- Libraries built on-top of Spark for popular use cases

Resilient Distributed Datasets (RDDs) (1/3)

- RDDs are distributed read-only collections of objects
 - partitioned and distributed across nodes
 - in-memory (intermediate results are not exchanged via disk like in MapReduce)
- Bulk operations (e.g. Map, Reduce, and Join) transform RDDs in parallel on workers



Resilient Distributed Datasets (RDDs) (2/3)

- RDDs are logical (lazy and ephemeral)
 - partitions of a dataset are materialized on demand when they are used
 - an RDD contains enough information to compute it starting from data in reliable storage
- Fault-tolerance through lineage
 - Affected partitions are recomputed on failures (but partitions can depend on all partitions of preceding RDDs → can be expensive)
 - No overhead in failure-free case

Resilient Distributed Datasets (RDDs) (3/3)

- Caching: users can give explicit hints that datasets should be kept in-memory
- Enables faster iterative jobs and interactive analysis

```
val data = spark.textFile(...).map(readPoint).cache()

var w = Vector.random(D)

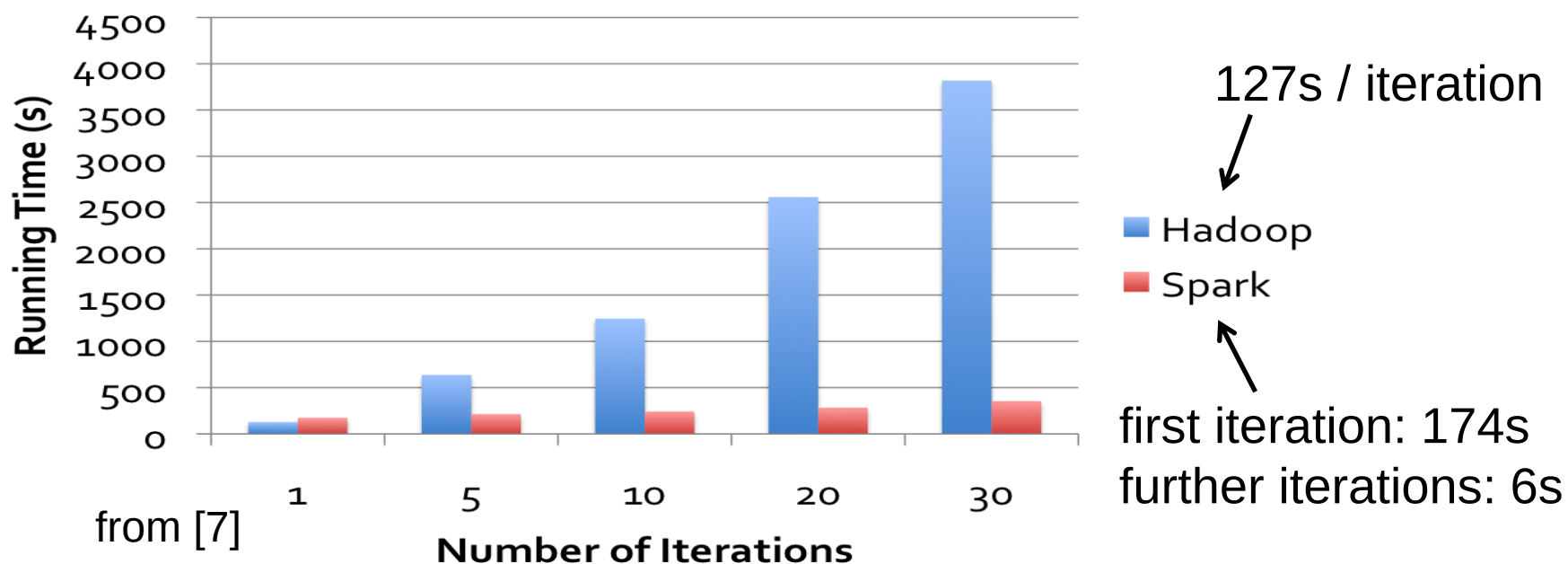
for (i <- 1 to ITERATIONS) {
  val gradient = data.map(p => {...}).reduce(_ + _)
  w -= gradient
}
```

from [7]

- Model data kept in-memory across iterations!

Apache Spark Performance

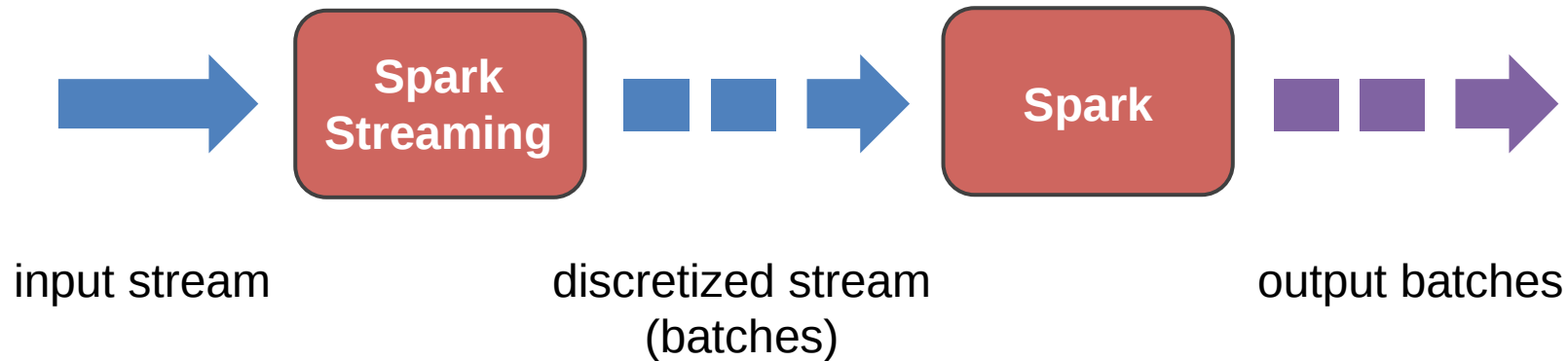
- Logistic Regression on 29 GB dataset using 20 “m1.xlarge” EC2 nodes



- In-memory caching makes subsequent iterations fast

Spark Streaming

- Spark processes continuous inputs in Microbatches
 - Arriving input is processed in small batches
 - Operations like aggregations and joins use these batches as their windows

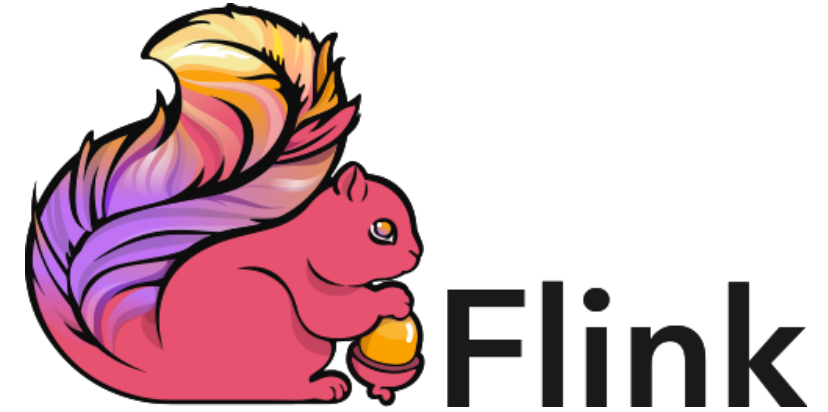


Overview

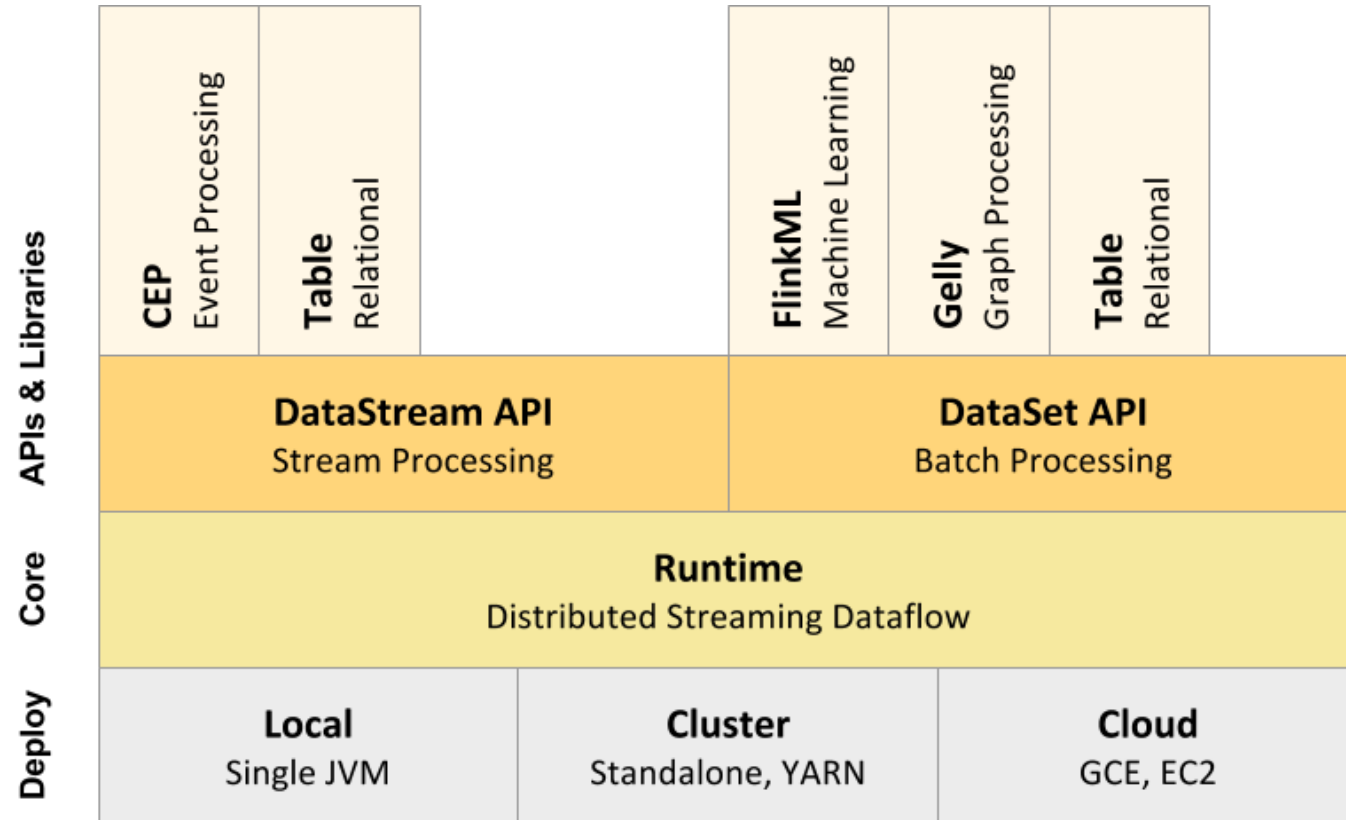
- Introduction
- Distributed Storage
 - GFS
 - HDFS
 - Bigtable
- Distributed Processing
 - MapReduce
 - Spark
 - **Flink**

Apache Flink[8]

- Open-source project that unifies batch and stream processing in one engine
- Started as joint research project from TU Berlin, HU Berlin, and HPI Potsdam
- Apache top-level project since beginning of 2015
 - Well-integrated in big data ecosystem
 - Active user and developer community worldwide

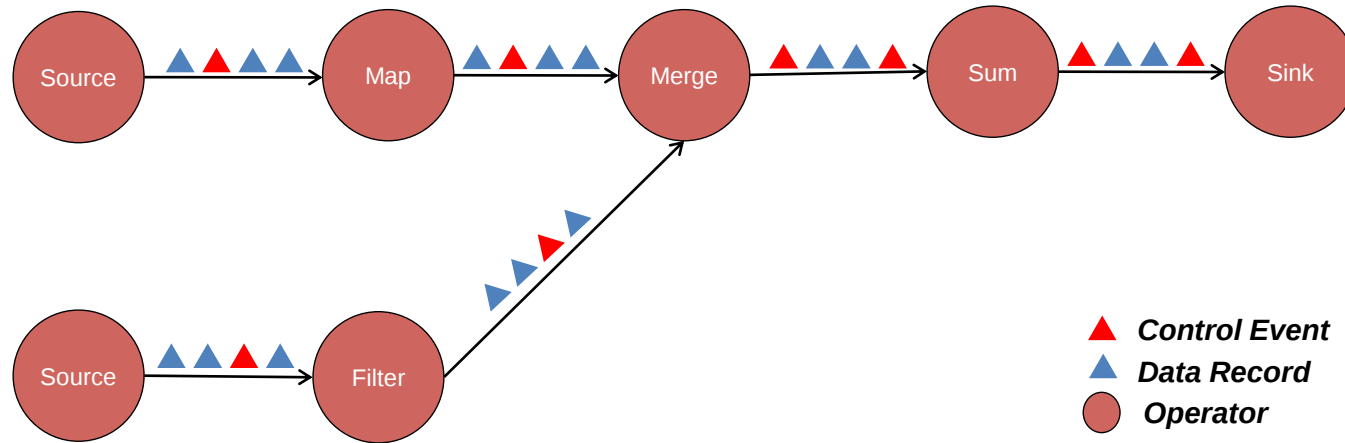


Apache Flink Stack



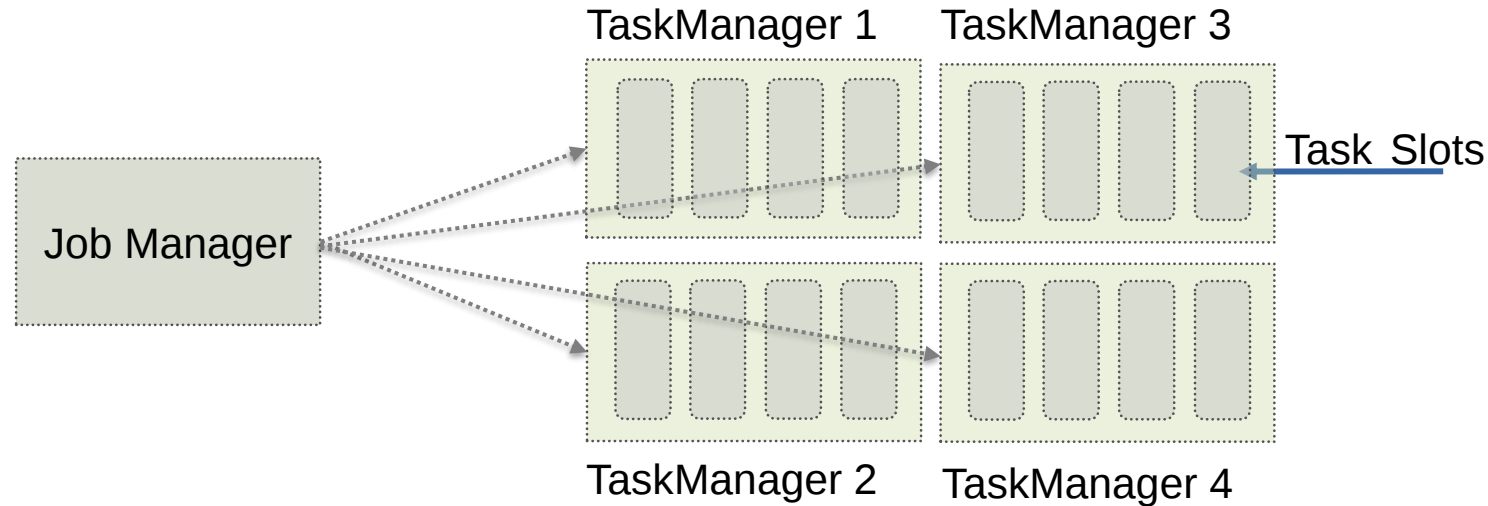
- Flink's core is a streaming dataflow engine that supports both batch and stream processing

Apache Flink Execution Model (1/3)



- A job consists of:
 - A directed acyclic graph (DAG) of operators and
 - Intermediate streams of data records and control events flowing through the DAG of operators
- Pipelined/Streaming engine

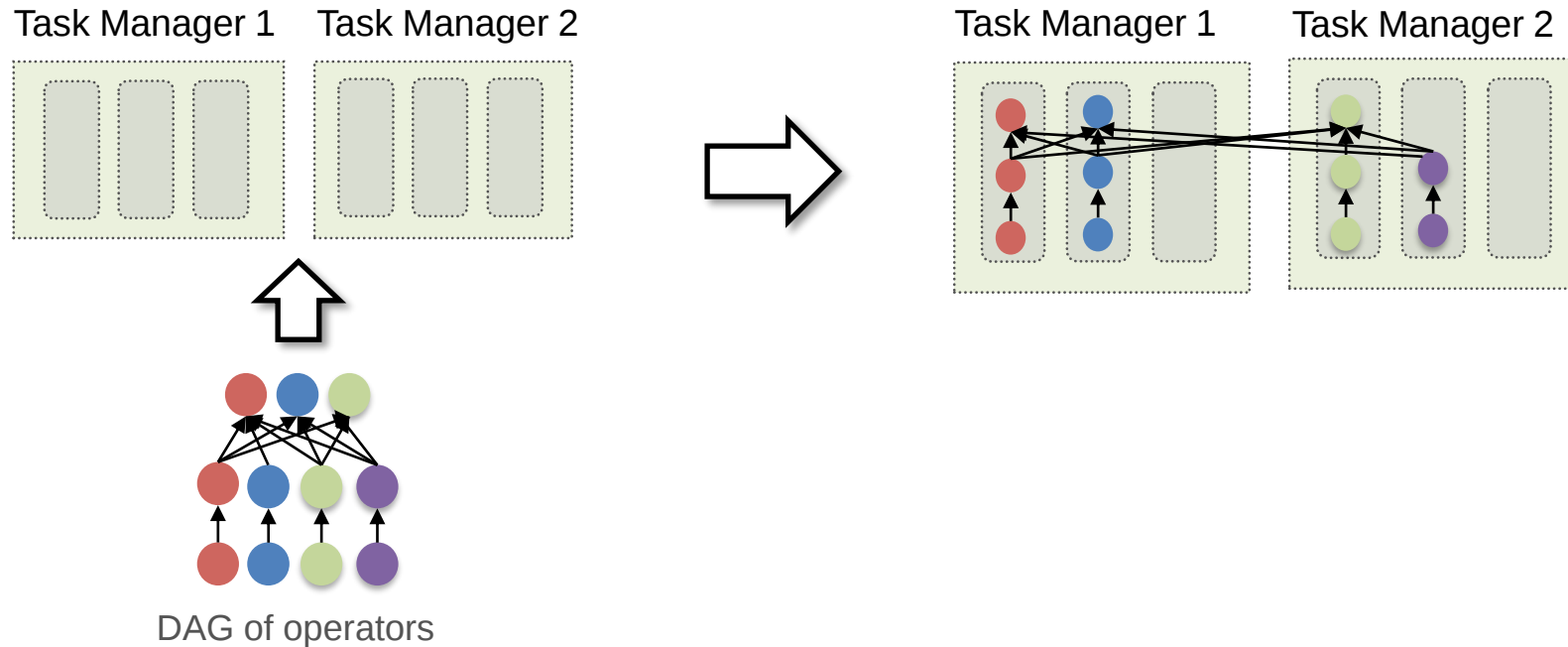
Apache Flink Execution Model (2/3)



- Follows a master-worker paradigm
 - Job Manager keeps track of all Task Managers and entire job execution
- Execution resources are defined through Task Slots
 - A Task Manager will have one or more Task Slots
 - Typically, there is one Task Slot per core

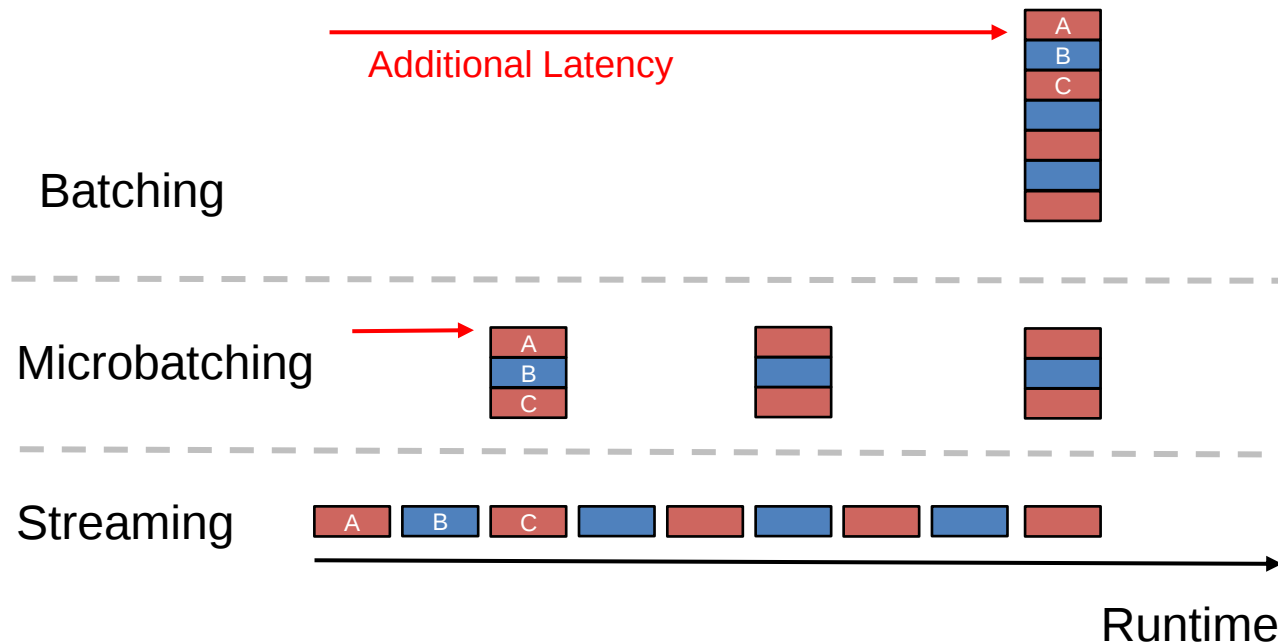
Apache Flink Execution Model (3/3)

- Deployment example on a cluster with 2 Task Managers with 3 slots each



- Complete DAG of operators are deployed distributed on all Task Managers
 - Task Slot executes a pipeline of tasks (operators)
 - Often execute successive operators concurrently

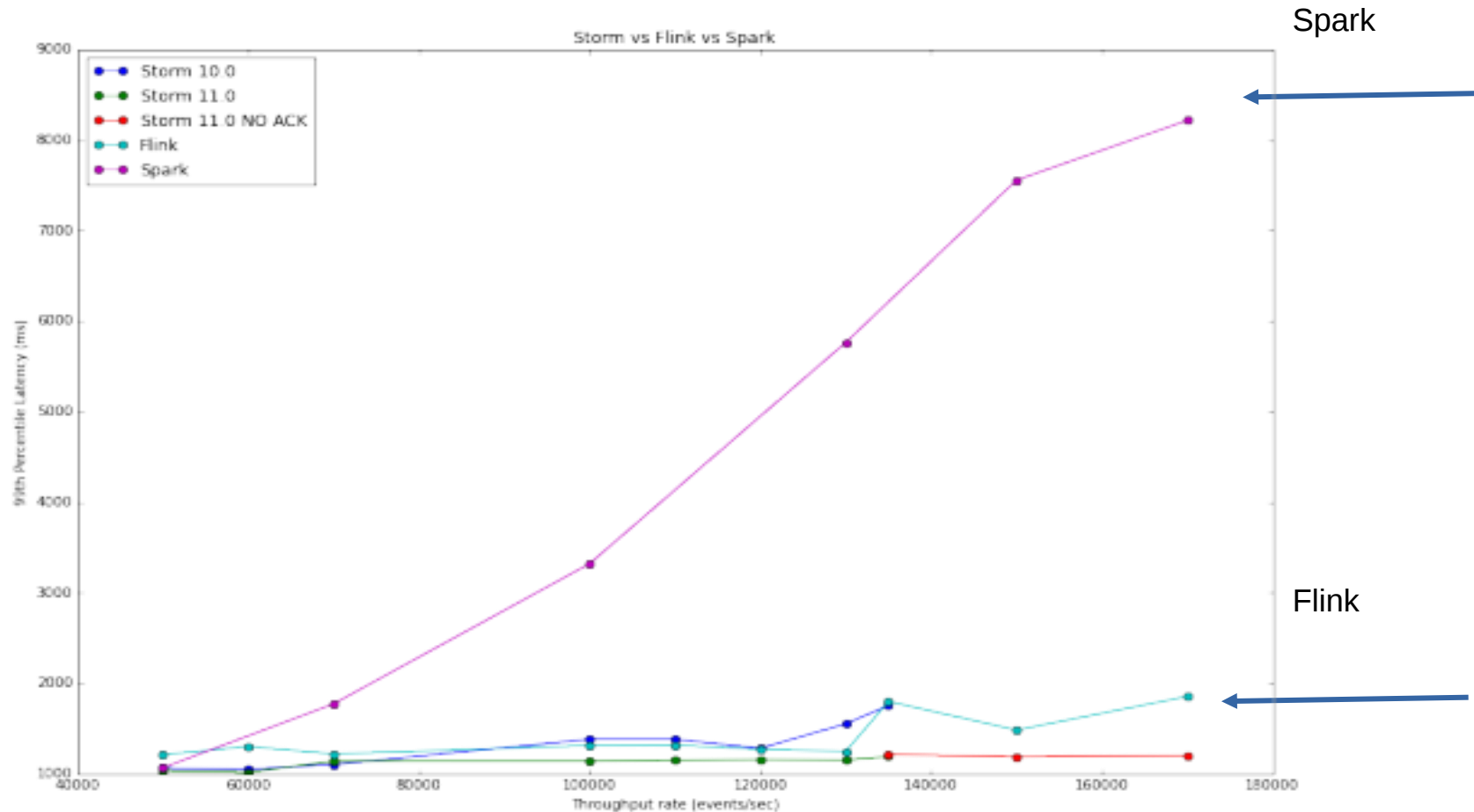
Performance Comparison of Streaming Approaches (1/2)



- Microbatches: Processing of small batches of tuples
- “Real” Streaming: Tuple-wise processing (lower latencies possible, but might cost throughput)

Performance Comparison of Streaming Approaches (2/2)

- Flink with lower latency bounds (y-axis): event processed as it becomes available



Overview

- Introduction
- Distributed Storage
 - GFS
 - HDFS
 - Bigtable
- Distributed Processing
 - MapReduce
 - Spark
 - Flink

Summary

- Major trends of the last two decades
 - Single processors stopped getting faster as quickly as before and storage prices decreased
 - More data is being recorded and stored
 - Scale-out over clusters of commodity resources and access to these resources via clouds (VMs / containers)
- New distributed systems to utilize clusters
 - Distributed storage systems for files and structured data
 - Distributed dataflow systems for search and relational processing, graph analysis, machine learning etc.

Literature and References

- Literature: M. Kleppmann, “Designing Data-Intensive Applications”, 2017, Chapter 10 and 11
- References:
 - [1] A. S. Das, M. Datar, A. Garg, and S. Rajaram, “Google News Personalization: Scalable Online Collaborative Filtering”, In Proceedings of the 16th International Conference on World Wide Web. WWW '07. ACM, 2007.
 - [2] J. Dean, S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters”, Communications of the ACM, 51 (1), 2008.
 - [3] R.E. Bryant, R.H. Katz, E.D. Lazowska, “Big-Data Computing: Creating Revolutionary Breakthroughs in Commerce, Science, and Society”, in Computing Research Initiatives for the 21st Century, Computing Research Association, 2008.
 - [4] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, “The Google File System”. in Proc. of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03), ACM, 2003.
 - [5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, R. E. Gruber: “Bigtable: A Distributed Storage System for Structured Data”, ACM Transactions on Computer Systems, 26 (2), 2008
 - [7] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker and I. Stoica. “Spark: Cluster Computing with Working Sets”, in Proc. of the 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'2010), 2010.
 - [8] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and Batch Processing in a Single Engine", in IEEE Data Engineering Bulletin iss. 36, no. 4, 2015.